

conference

proceedings

FREENIX Track
2000 USENIX Annual
Technical Conference

San Diego, CA, USA
June 18–23, 2000

Sponsored by

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: <http://www.usenix.org>

The price is \$22 for members and \$30 for nonmembers.
Outside the U.S.A. and Canada, please add
\$18 per copy for postage (via air printed matter).

Past FREENIX Proceedings

FREENIX '99	1999	Monterey, CA	\$22/30
-------------	------	--------------	---------

© 2000 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-23-5

Printed in the United States of America on 50% recycled paper, 10–15% post consumer waste.

Jeff Mogul

USENIX Association

**Proceedings of the
FREENIX Track**

2000 USENIX Annual Technical Conference

**June 18–23, 2000
San Diego, California, USA**

Program Organizers

Program Chair

Marshall Kirk McKusick, *Author & Consultant*

Program Committee

Clem Cole, *Compaq*

Chris Demetriou, *SiByte, Inc.*

David Greenman, *The FreeBSD Project*

Miguel de Icaza, *Helix Code, Inc.*

Niels Provos, *University of Michigan*

Theodore Ts'o, *VA Linux Systems*

Stephen C. Tweedie, *Red Hat, Inc.*

Victor Yodaiken, *FSMLabs*

The USENIX Association Staff

FREENIX Track

2000 USENIX Annual Technical Conference

June 18–23, 2000

San Diego, California, USA

Index of Authors	vi
Message from the Program Chair	vii

Wednesday, June 21

Storage Systems

Session Chair: Marshall Kirk McKusick, Author & Consultant

Swarm: A Log-Structured Storage System for Linux	1
<i>Ian Murdock and John H. Hartman, University of Arizona</i>	

DMFS—A Data Migration File System for NetBSD	11
<i>William Studenmund, Veridian MRJ Technology Solutions</i>	

A 3-Tier RAID Storage System with RAID1, RAID5, and Compressed RAID5 for Linux	21
<i>K. Gopinath, Nitin Muppalaneni, N. Suresh Kumar, and Pankaj Risbood, Indian Institute of Science, Bangalore</i>	

Network System Administration

Session Chair: Victor Yodaiken, FSMLabs

Extending Internet Services via LDAP	35
<i>James E. Dutton, Southern Illinois University</i>	

MOSIX: How Linux Clusters Solve Real-World Problems	49
<i>Steve McClure and Richard Wheeler, EMC² Corp.</i>	

Webmin: A Web-Based System Administration Tool for UNIX	57
<i>Jamie Cameron, Caldera Systems</i>	

File Systems

Session Chair: Ted Ts'o, VA Linux Systems

Porting the SGI XFS File System to Linux	65
<i>Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan, Ken McDonell, Ted Kline, Brian Gaffey, and Rajagopal Ananthanarayanan, SGI</i>	

LinLogFS—A Log-Structured File System for Linux	77
<i>Christian Czeatzke, xS+S; M. Anton Ertl, TU Wien</i>	

UNIX File System Extensions in the GNOME Environment	89
<i>Ettore Perazzoli, Helix Code, Inc.</i>	

Thursday, June 22

Sockets

Session Chair: David Greenman, The FreeBSD Project

Protocol Independence Using the Sockets API99
Craig Metz, University of Virginia

Scalable Network I/O in Linux109
Niels Provos, University of Michigan; Chuck Lever, Sun-Netscape Alliance

Accept() Scalability in Linux121
Stephen P. Molloy, University of Michigan; Chuck Lever, Sun-Netscape Alliance

Network Publishing

Session Chair: Chris Demetriou, AT&T Labs

Permanent Web Publishing129
David S. H. Rosenthal, Sun Microsystems Laboratories; Vicky Reich, Stanford University Libraries

The Globe Distribution Network141
A. Bakker, E. Amade, and G. Ballintijn, Vrije Universiteit Amsterdam; I. Kuz, Delft University of Technology; P. Verkaik, I. van der Wijk, M. van Steen, and A. S. Tanenbaum, Vrije Universiteit Amsterdam

Open Information Pools153
Johan Pouwelse, Delft University of Technology

X11 and User Interfaces

Session Chair: Miguel de Icaza, Helix Code, Inc.

The GNOME Canvas: A Generic Engine for Structured Graphics165
Federico Mena-Quintero, Helix Code, Inc.; Raph Levien, Code Art Studio

Efficiently Scheduling X Clients175
Keith Packard, SuSE Inc.

The AT&T AST OpenSource Software Collection187
Glenn S. Fowler, David G. Korn, Stephen S. North, and Kiem-Phong Vo, AT&T Laboratories—Research

Friday, June 23

Security

Session Chair: Niels Provos, University of Michigan

Implementing Internet Key Exchange (IKE)201
Niklas Hallqvist, Applitron Datasystem AB; Angelos D. Keromytis, University of Pennsylvania

Transparent Network Security Policy Enforcement215
Angelos D. Keromytis, University of Pennsylvania; Jason L. Wright, Network Security Technologies, Inc. (NETSEC)

Safety Checking of Kernel Extensions227
Craig Metz, University of Virginia

Cool Stuff

Session Chair: Clem Cole, Compaq

An Operating System in Java for the Lego Mindstorms RCX Microcontroller235
Pekka Nikander, Helsinki University of Technology

LAP: A Little Language for OS Emulation249
Donn M. Seeley, Berkeley Software Design, Inc.

Traffic Data Repository at the WIDE Project263
Kenjiro Cho, Sony CSL; Koushirou Mitsuya, Keio University; Akira Kato, University of Tokyo

Short Topics

Session Chair: Stephen C. Tweedie, Red Hat, Inc.

JEmacs—The Java/Scheme-based Emacs271
Per Bothner

A New Rendering Model for X279
Keith Packard, SuSE, Inc.

UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD285
Chuck Silvers, The NetBSD Project

Mbuf Issues in 4.4BSD IPv6 Support—Experiences from KAME IPv6/IPsec Implementation291
Jun-ichiro itojun Hagino, Internet Initiative Japan Inc.

Malloc() Performance in a Multithreaded Linux Environment301
Chuck Lever and David Boreham, Sun-Netscape Alliance

Index of Authors

Amade, E.	141	McClure, Steve	49
Ananthanarayanan, Rajagopal	65	McDonell, Ken	65
Bakker, A.	141	Mena-Quintero, Federico	165
Ballintijn, G.	141	Metz, Craig	227, 99
Boreham, David	301	Mitsuya, Koushirou	263
Bothner, Per	271	Molloy, Stephen P.	121
Cameron, Jamie	57	Mostek, Jim	65
Cattelan, Russell	65	Muppalaneni, Nitin	21
Cho, Kenjiro	263	Murdock, Ian	1
Czezatke, Christian	77	Nikander, Pekka	235
Dutton, James E.	35	North, Stephen S.	187
Earl, Bill	65	Packard, Keith	175, 279
Ertl, M. Anton	77	Perazzoli, Ettore	89
Fowler, Glenn S.	187	Pouwelse, Johan	153
Gaffey, Brian	65	Provos, Niels	109
Gopinath, K.	21	Reich, Vicky	129
Hagino, Jun-ichiro itojun	291	Risbood, Pankaj	21
Hallqvist, Niklas	201	Rosenthal, David S. H.	129
Hartman, John H.	1	Seeley, Donn M.	249
Kato, Akira	263	Silvers, Chuck	285
Keromytis, Angelos D.	201, 215	Steen, M. van	141
Kline, Ted	65	Studenmund, William	11
Korn, David G.	187	Tanenbaum, A. S.	141
Kumar, N. Suresh	21	Verkaik, P.	141
Kuz, I.	141	Vo, Kiem-Phong	187
Lever, Chuck	109, 121, 301	Wheeler, Richard	49
Levien, Raph	165	Wijk, I. van der	141
Levine, Steven	65	Wright, Jason L.	215
Lord, Steve	65		

Message from the Program Chair

This year represents the third time that the FREENIX track has been organized, and, as in the past two years, enthusiasm and interest have again increased dramatically. It has been my great pleasure to work with the FREENIX Program Committee in organizing this year's FREENIX track. The program committee put in many hours, which started by having every member of the committee read and provide written feedback on every submitted paper. The committee members then trekked through a January blizzard to meet for two days in Newark, New Jersey.

Following the meeting, each accepted paper was assigned a program committee member as shepherd. The role of the shepherds was to help their authors prepare their papers for publication. Our first goal was to put together a program that included work from both Linux and BSD developers, as well as others who are developing applications in the Open Source world. The diversity of papers in this program show that we very successfully met that goal. Our other goal was to seek out new developers who had little or no publication experience and for whom trying to reach the standards of the refereed track was too daunting. Through the hard work of our authors and their shepherds, we have reached a new FREENIX milestone, in which every FREENIX presentation has a complete paper in the proceedings.

I hope that you will come spend some time in the FREENIX sessions learning about the exciting work going on in the Open Source world.

Marshall Kirk McKusick, FREENIX Program Chair

Swarm: A Log-Structured Storage System for Linux

Ian Murdock

Department of Computer Science

The University of Arizona

imurdock@cs.arizona.edu

John H. Hartman

Department of Computer Science

The University of Arizona

jhh@cs.arizona.edu

Abstract

Swarm [3] is a storage system for Linux that provides scalable, reliable, and cost-effective data storage. At its lowest level, Swarm implements a log-structured interface to a cluster of storage devices. Above the log, Swarm provides an infrastructure that allows high-level abstractions and functionality to be implemented easily and efficiently. This paper describes the design and implementation of Swarm, paying particular attention to the Swarm infrastructure and how it has been used to construct two storage systems: Sting, a log-structured file system for Linux, and ext2fs/Swarm, a Swarm-based version of the Linux ext2 file system that runs unmodified above a block device compatibility layer. The paper concludes with a discussion of our experiences using Linux as a platform for research.

1 Introduction

In Linux, file systems interface with applications through an abstraction layer called the Virtual Filesystem Switch (VFS). The VFS separates file system interface from implementation, allowing many different file systems to coexist in a single file system namespace. The VFS implements functionality common to all file systems, including the system calls that access or modify file system data and metadata (e.g., `read` and `mkdir`). To perform file-system-specific operations, the VFS vectors control to lower-level file system implementations when appropriate (e.g., when a block needs to be read from

disk, or when an entry needs to be added to a directory). Below the VFS, file systems access storage indirectly through a caching subsystem. For example, disk-based file systems communicate with the underlying disks through a buffer cache, which provides a level of indirection between disks and the file systems stored on them.

In general, high-level abstractions such as those provided by file systems are tightly coupled with the low-level storage devices on which they are implemented, making it difficult to extend or configure the storage system. For example, since the ext2 file system interfaces with disks through the buffer cache, it is not possible to run ext2 above a storage device that is not block-oriented. Furthermore, the high-level abstractions themselves are often tightly coupled, providing a single large feature set that is difficult to change without directly modifying the file system. For example, ext2 implements the standard UNIX file system functionality and interface. In general, to extend the functionality of ext2 (e.g., to add support for transparent compression of file data), it is necessary to modify ext2 directly to support the desired features.

To address this inherent inflexibility, several projects have created extensible storage systems. One group of projects focuses on stacking vnodes [5, 11]. In many versions of UNIX, a vnode represents an object in the file system namespace, such as a file, directory, or named pipe. In the context of Linux, which does not support vnodes, the same effect may be achieved by layering at the VFS level, allowing new functionality to be interposed between layers. For example, to add compression to

a file system, a file compression layer is interposed between the file system and the buffer cache that compresses files as they are written, uncompresses files as they are read, and manages all the details required to make this transparent (e.g., making sure cache blocks are managed properly). Thus, VFS layering allows certain functionality to be added to file systems without having to modify the file systems themselves.

Other projects attempt to provide flexibility at the bottom end. For example, Linux supports transparently-compressed block devices, providing support for compression of file system data and metadata at the block level, and software RAID, which combines several physical block devices into one virtual block device that looks and acts like a RAID. This kind of support allows high-level abstractions to escape the confines of a particular storage abstraction to a limited extent without requiring modification of the higher levels.

The problem with these approaches is that they only allow changes in implementation; they do not allow changes in interface. At the top end, VFS layering does not allow extensions to alter the file-oriented interface the VFS provides; this limits the expressibility of VFS layers to functionality that matches the file abstraction and interface. For example, file system filters, like transparent compression and encryption, fit very nicely into the VFS framework, but other kinds of extensions that diverge from the file abstraction and interface are more difficult to implement, such as integrated support for database-like functionality (e.g., transactions). At the bottom end, extensions like transparent compression of block devices and software RAID allow some file systems to provide extended functionality, but only those that support the particular storage abstraction being extended (in this case, those file systems that run above block devices).

Swarm [3] is a storage system that attempts to address the problem of inflexibility in storage systems by providing a configurable and extensible infrastructure that may be used to build high-level storage abstractions and functionality. At its lowest level, Swarm provides a log-structured interface to a cluster of storage devices that act as repositories for fixed-sized pieces of the log called *fragments*. Because the storage devices have relatively simple functionality, they are easily implemented using inexpensive commodity hardware or network-attached disks [2]. Each storage device is optimized

for cost-performance and aggregated to provide the desired absolute performance.

Swarm clients use a *striped log* abstraction [4] to store data on the storage devices. This abstraction simplifies storage allocation, improves file access performance, balances server loads, provides fault-tolerance through computed redundancy, and simplifies crash recovery. Each Swarm client creates its own log, appending new data to the log and forming the log into fragments that are striped across the storage devices. The parity of log fragments is computed and stored along with them, allowing missing portions of the log to be reconstructed when a storage device fails. Since each client maintains its own log and parity, the clients may act independently, resulting in improved scalability, reliability, and performance over centralized file servers.

Swarm is a storage system, not a file system, because it can be configured to support a variety of storage abstractions and access protocols. For example, a Swarm cluster could simultaneously support Sun's Network File System (NFS) [10], HTTP, a parallel file system, and a specialized database interface. Swarm accomplishes this by decoupling high-level abstractions and functionality from low-level storage. Rather than providing these abstractions directly, Swarm provides an infrastructure that allows high-level functionality to be implemented above the underlying log abstraction easily and efficiently. This infrastructure is based on layered modules that can be combined together to implement the desired functionality. Each layer can augment, extend, or hide the functionality of the layers below it. For example, an atomicity service can layer above the log, providing atomicity across multiple log operations. In turn, a logical disk service can layer above this extended log abstraction, providing a disk-like interface to the log and hiding its append-only nature. This is in contrast to VFS or vnode layering, in which there is a uniform interface across all layers.

Swarm has been under development at the University of Arizona for the past two years. We have implemented the Swarm infrastructure in both a user-level library and the Linux kernel (versions 2.0 and 2.2), and we have used this infrastructure to implement two storage systems, Sting, a log-structured file system for Linux, and ext2fs/Swarm, a Swarm-based version of the Linux ext2 file system that runs unmodified above a block device compatibility layer.

2 Swarm infrastructure

Swarm provides an infrastructure for building storage services, allowing applications to tailor the storage system to their exact needs. Although this means that many different storage abstractions and communication protocols are possible, Swarm-based storage systems typically store data in a *striped log* storage abstraction and use a storage-optimized protocol for transferring data between client and server.

In the striped log abstraction, each client forms data into an append-only log, much like a log-structured file system [9]. The log is then divided into fixed-sized pieces called *fragments* that are striped across the storage devices. Each fragment is identified by a 64-bit integer called a *fragment identifier (FID)*. Fragments may be given arbitrary FIDs, allowing higher levels to construct multi-device fragment address spaces. As the log is written, the parity of the fragments is computed and stored, allowing missing fragments to be reconstructed should a storage device fail. A collection of fragments and its associated parity fragment is called a *stripe*, and the collection of devices they span is called a *stripe group*.

The striped log abstraction is central to Swarm's high-performance and relatively simple implementation. The log batches together many small writes by applications into large, fragment-sized writes to the storage devices, and stripes across the devices to improve concurrency. Computing parity across log fragments, rather than file blocks, decouples the parity overhead from file sizes, and eliminates the need for updating parity when files are modified, since log fragments are immutable. Finally, since each client writes its own log, clients can store and access data on the servers without coordination between the clients or the servers.

Swarm is implemented as a collection of modules that may be layered to build storage systems in much the same way that protocols may be layered to build network communications subsystems [6]. Each module in Swarm implements a storage service that communicates with the lower levels of the storage system through a well-defined interface, and exports its own well-defined interface to higher levels. Storage systems are constructed by layering the appropriate modules such that all interfaces between modules are compatible. This section describes the basic modules that are used to construct storage systems in Swarm.

2.1 Disk

As in most storage systems, the disk is the primary storage device in Swarm. Swarm accesses disks through the *disk layer*. The disk layer exports a simple, fragment-oriented interface that allows the layers above to read, write, and delete fragments. Fragment writes are atomic; if the system crashes before the write completes, the disk state is "rolled back" to the state it was in prior to the write.

The disk layer operates by dividing the disk into fragment-sized pieces and translating fragment requests into disk requests. The mappings from fragment identifiers to disk addresses are stored in an on-disk *fragment map* that is stored in the middle of the disk to reduce access time. The disk contains two copies of the fragment map, each with a trailing timestamp, to permit recovery when the system crashes while a fragment map write is in progress. As an optimization, the disk layer only writes out the fragment map periodically, saving two additional seeks and a disk write on most fragment operations.

The fragment map is not written to disk each time it is updated, so the disk layer must be able to make it consistent again after crashes, or fragments written after the last fragment map write will be lost. To address this problem, the disk layer borrows a trick from the Zebra storage server [4]. It includes a header with each fragment that contains enough information for the fragment map to be updated appropriately at recovery time. To allow crash recovery to proceed without having to scan the entire disk, the disk layer preallocates the next set of fragments to write before it writes the fragment map and stores their locations in the fragment map. At recovery time, the disk layer need only examine the fragments preallocated in the fragment map.

2.2 Network-attached storage

Swarm also supports the use of network-attached storage devices. Swarm provides a network transparency layer that has the same interface as the disk layer, allowing locally-attached and network-attached storage devices to be used interchangeably. The network transparency layer accepts fragment operations from the layers above it, sends them across the network to the appropriate device, and re-

turns the result of the operation to the caller transparently.

In Swarm, network-attached storage devices are called *storage servers*. The storage servers are essentially enhanced disk appliances running on commodity hardware, and provide the same fragment-oriented interface as disks, with added support for security. The storage server security mechanism consists of access control lists that may be applied to arbitrary byte ranges within a fragment. The lists give a client fine-grained control over which clients can access its data, without limiting the ways in which fragments may be shared.

2.3 Striper

The *striper* layer is responsible for striping fragments across the storage devices. It exports a fragment-oriented interface to the higher layers. As fragments are written, the striper forms the fragments into stripes and writes them to the appropriate storage devices in parallel. To provide flow control between the striper and the storage devices, the striper maintains a queue of fragments to be written for each device; the striper puts fragments into these queues, and the storage layers take them out and store them on the appropriate device. Fragments are written to disk one-at-a-time, but with a storage server, the network layer transfers the next fragment to the server while the previous one is being written to disk; this keeps both the storage server's disk and the network busy, so that as soon as one fragment has been written to disk, the server can immediately begin writing another fragment.

2.4 Parity

The *parity layer* implements the standard parity mechanism used by RAID [8]. The parity layer sits above the striper and provides a compatible interface, allowing the striper and parity layer to be used interchangeably. One difficulty that arises from Swarm's support for stripe groups is determining which fragments constitute the remaining fragments of a stripe during reconstruction, and on which devices they are stored. Swarm solves this problem by storing stripe group information in each fragment of a stripe, and numbering the fragments in the same stripe consecutively. If fragment N needs to be reconstructed, then either fragment N-1 or fragment

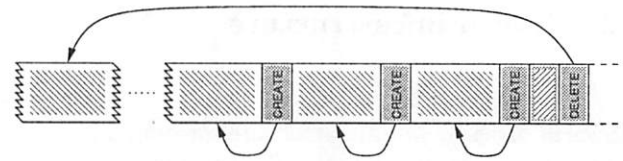


Figure 1: The light objects are blocks, and the dark objects are records. Each CREATE record indicates the creation of a block, and each DELETE record indicates a deletion; the arrows show which block is affected by each record and represent references visible to the log layer. Note that the contents of the blocks themselves are uninterpreted by the log layer.

N+1 is in the same stripe. The client queries all the storage devices until it finds either fragment N-1 or N+1. For locally-attached disks, the client uses configuration information to find all the disks; for storage servers, the client simply broadcasts to find the desired fragments. Broadcast is used because it is simple and makes Swarm self-hosting—no additional mechanism is needed to distribute stripe group and storage server information reliably to all clients.

2.5 Striped log

Above the fragment-oriented interfaces provided by the storage, striper, and parity layers is the *log layer*. The log layer implements the striped log storage abstraction and corresponding interface, forming data written by higher levels into an append-only log and striping the log across the underlying storage devices. The layers above the log are called *storage services* (*services* for short) and are responsible for implementing high-level storage abstractions and functionality. The log layer's main function is to multiplex the underlying storage devices among multiple services, allowing storage system resources to be shared easily and efficiently.

The log is a conceptually infinite, ordered stream of *blocks* and *records* (Figure 1). It is append-only: blocks and records are written to the end of the log and are immutable. Block contents are service-defined and are not interpreted by the log layer. For example, a file system would use blocks not only to store file data, but also inodes, directories, and other file system metadata. Once written, blocks persist until explicitly deleted, though their physical locations in the log may change as a result of cleaning

or other reorganization.

New blocks are always appended to the end of the log, allowing the log layer to batch together small writes into fragments that may be efficiently written to the storage devices. As fragments are filled, the log layer passes them down to the striper for storage. Once written, a block may be accessed given its *log address*, which consists of the FID of the fragment in which it is stored and its offset within the containing fragment. Given a block's log address and length, the log layer retrieves the block from the appropriate storage device and returns it to the calling service. When a service stores a block in the log, the log layer responds with its log address so that the service may update its metadata appropriately.

Records are used to recover from client crashes. A crash causes the log to end abruptly, potentially leaving a service's data structures in the log inconsistent. The service repairs these inconsistencies by storing state information in records during normal operation, and re-applying the effect of the records after a crash. For example, a file system might append records to the log as it performs high-level operations that involve changing several data structures (e.g., as happens during file creation and deletion). During replay, these records allow the file system to easily redo (or undo) the high-level operations. Records are implicitly deleted by *checkpoints*, special records that denote consistent states. The log layer guarantees atomicity of record writes and preserves the order of records in the log, so that services are guaranteed to replay them in the correct order.

2.6 Stripe cleaner

As in LFS, Swarm uses a *cleaner* to periodically compress free space in the log to make room for new stripes [9]. In Swarm, the cleaner is implemented as a layer above the log, hiding the log's finite capacity from higher-level services. The cleaner service monitors the blocks and records written to the log, allowing it to track which portions of the log are unused. A block is cleaned by re-appending it to the log, which changes its address and requires the service that wrote it to update its metadata accordingly. When a block is cleaned, the cleaner notifies the service that created it that the block has moved. The notification contains the old and new addresses of the block, as well as the block's creation record.

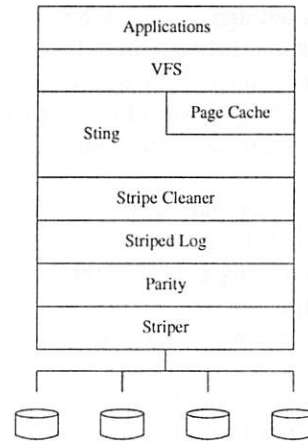


Figure 2: This figure shows the Swarm module configuration for Sting. The storage devices at the bottom layer may be either locally-attached or network-attached disks, and the number of devices is configurable.

The creation record contains service-specific information that makes it easier for the service to update its metadata. For example, the creation record for a file block might contain the inode number of the block's file and the block's offset. The cleaner is also responsible for free space management, enforcing quotas on higher-level services, reserving the appropriate number of stripes so that cleaning is always able to proceed, and initiating cleaning to make room for new stripes in the event there are no free stripes.

3 Swarm storage systems

We have implemented the Swarm infrastructure described in the previous section in both a user-level library and the Linux kernel, and we have used this infrastructure to implement two storage systems. This section describes our two demonstration storage systems, Sting, a log-structured file system for Linux, and ext2fs/Swarm, a Swarm-based version of the Linux ext2 file system that runs unmodified above a block device compatibility layer.

3.1 Sting

Sting is a log-structured file system for Linux that is based on Swarm. It is similar to Sprite LFS [9],

although it is smaller and simpler because the underlying Swarm infrastructure deals transparently with log management and storage, cleaning, and other LFS tasks. As with all other Linux-based file systems, Sting interacts with applications through the Linux Virtual Filesystem Switch (VFS). Thus, Linux applications may run above Sting as they would any other Linux file system. However, unlike other Linux-based file systems, Sting accesses storage through the striped log abstraction rather than a block device (see Figure 2).

Sting stores all of its data and metadata in blocks, and uses many of the same data structures as LFS. Files are indexed using the standard UNIX inode mechanism. Directories are simply files mapping names to inode numbers. Inodes are located via the *inode map*, or *imap*, which is a special file that stores the current location of each inode. A similar data structure is not needed in most file systems because inodes are stored at fixed locations on the disk and modified in place. In Sting, when an inode is modified or cleaned, a new version of the inode is written to the end of the log, causing the inode's address to change periodically.

Sting uses records to recover its state after a crash. For example, when a file is created, Sting writes a record to this effect to the log, so that it may easily recreate the file after a crash. This mechanism is similar to that used by journaling file systems. Without Swarm's record mechanism, Sting would be forced to write out the affected metadata when a file is created, and write it out in a particular order so that a file system integrity checker (e.g., *fsck*) can rectify an inconsistency after a crash. In Swarm, services may summarize changes in records without actually making them, and they are freed from having to do their own ordering because Swarm guarantees that records are properly ordered by time of creation during recovery.

Sting runs above Swarm's striped log, not a block device, so it is unable to use the buffer cache used by disk-based file systems for buffering writes and caching metadata. Rather than adding our own specialized buffering mechanism to Sting, we modified the page cache to support tracking and writing back dirty pages so that Sting could use it as its primary file cache for both reads and writes. With our changes, pages may be modified and marked "dirty" for later writing. When a page is marked dirty, it is added to a list of dirty pages. Then, during each run of *update*, the list of dirty pages is

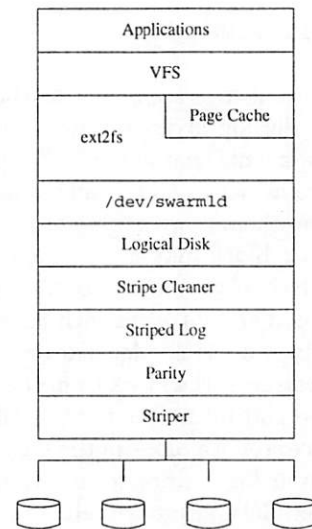


Figure 3: This figure shows the Swarm module configuration for ext2fs/Swarm. The stack below the file system is identical to that of Figure 2, with the introduction of two compatibility layers. As with Sting, the storage devices at the bottom layer may be either locally-attached or network-attached disks, and the number of devices is configurable.

traversed. If a page is older than a certain threshold age (currently twenty-five seconds), it is written via a VFS method called *writepage*. In addition, Sting uses the page cache to cache indirect blocks, directory blocks, symbolic links, and the inode map. Our modifications to the page cache were small but allowed us to more easily implement a file system that does not run above a block device.¹

Sting accesses storage through Swarm, allowing it run on either locally-attached disks or on a Swarm cluster as a network file system (or a combination of the two). In addition, it may also take full advantage of Swarm's striping capabilities transparently. It does not yet support file sharing between clients. We are in the process of implementing a locking service that will allow clients to synchronize file accesses, allowing us to easily modify Sting to be used as a distributed file system.

3.2 Ext2fs/Swarm

Ext2fs/Swarm is a version of the Linux ext2 file system that runs unmodified above Swarm. This is possible through the use of a special logical disk service that provides a disk-like interface above Swarm's striped log, and a Linux block device driver that translates Linux block device requests into Swarm logical disk requests (see Figure 3). The block device driver exports the usual block device interface via the `/dev/swarmld` device file, which may be read, written, or mounted like any other block device. Thus, via the logical disk service and `/dev/swarmld`, Swarm appears to Linux to be just an ordinary block device.

The Swarm logical disk is similar to the MIT Logical Disk [1], but it is much simpler because it is implemented as a service above Swarm. It hides the append-only nature of the striped log, providing the illusion that higher-level services are running above a disk. As with real disks, blocks in a logical disk are written to fixed addresses that do not change over time, and that can be overwritten in place. This provides a much more convenient abstraction for some services than the append-only log, in which services must always append blocks to the end of the log and block addresses change as the result of being cleaned or overwritten.

The Swarm logical disk service's primary function is to maintain a mapping from a logical disk's finite address space to the log's infinite address space. This mapping is maintained in main memory, and is periodically written to the log by the logical disk service. In addition to updating the mapping table during normal operation, the logical disk service intercepts change-of-address notifications from the stripe cleaner as blocks are cleaned and updates the mapping table transparently.

Although the logical disk is a general-purpose storage service that may be used to implement any storage system, it serves as an excellent compatibility layer that allows existing file systems that run above a disk to run unmodified above Swarm. Using `/dev/swarmld`, ext2fs (and all other disk-based Linux file systems) may run above Swarm unmodified. As read requests and write requests are generated by applications, they are passed to the device

¹Many of these shortcomings have been addressed in the latest development version of Linux, 2.3. Our development efforts were under Linux 2.0 and Linux 2.2.

driver; in turn, the requests are passed to the logical disk, which performs the appropriate operation on the striped log. In addition to striping, improved performance on small writes, and other benefits provided by Swarm, ext2fs/Swarm can be configured to run on a Swarm cluster as a network file system. Note, however, that because we run ext2fs unmodified, its concurrency aspects are unchanged, so only one client at a time may have write access to any given instance of it.

4 Experiences using Linux for research

Swarm has been under development in the Department of Computer Science at the University of Arizona for the past two years. Early on, we decided to base Swarm on Linux because Linux had a large and rapidly-growing user base, and we wanted to build a storage system that people could use for day-to-day data storage. As with most decisions, ours was not without a downside, and we have learned some important lessons about Linux over the past few years that may prove useful to fellow researchers who are considering using Linux as a platform for research. We hope these observations will be equally useful to Linux developers, and that they will help make Linux an even better platform for research and development.

The lack of documentation is one of the biggest limitations of Linux to the uninitiated. There is very little documentation on the internal workings of Linux, and what little exists tends to become outdated quickly. The code is often the best (and sometimes the only) point of reference on how a particular part of Linux works. Unfortunately, the functionality of a piece of code is not always obvious at first glance, as comments are sparse in many places, and the code is often highly optimized and thus frequently difficult to understand at first glance.

Fortunately, Linux has a large and friendly development community that is normally more than happy to answer questions (as long as the asker has done his homework first), and the Internet serves as an invaluable archival reference for finding out if someone else has asked the same questions in the past (they almost certainly have). Still, Linux would do well to improve its documentation efforts. Since the code evolves so rapidly, any successful documenta-

tion mechanism must be somehow tied to the code itself (e.g., automatically generated from comments in the code), to prevent divergence of documentation from documented functionality.

Another limitation of Linux is its rather spartan development environment. Linux (at least the x86 version) does not include a kernel debugger, which we consider to be an essential part of any operating system development environment. Rather than using an interactive debugger, Linux developers prefer to rely on `printk` to debug new code, and use textual crash (“oops”) dumps of registers, assembly code, and other low-level system state when something goes wrong. Fortunately, there are patches available that allow the kernel to be debugged interactively. Linux is also light on diagnostic facilities. We often found ourselves having to manually poke around inside system data structures using the debugger or `printk` to gain insight into the source of a bug when something had gone wrong. A clean interface to displaying and analyzing system state, accessible from the debugger, is another integral part of any operating system development environment.

Finally, Linux does not always follow good software engineering practices [7]. For example, much of the code we worked with seemed optimized unnecessarily, usually at the expense of code clarity. Operating system code should only be optimized when doing so has a clear, *quantified* impact on *overall* performance. Most operating system code is not performance critical and has little or no effect on overall performance; such code should be structured for the human reader rather than the computer, to make the code easier to understand and maintain, to make bugs less likely to be introduced when the code is modified, and to make bugs that are introduced easier to find and fix. Furthermore, the abstraction boundaries in Linux are not always obeyed or even well-defined. We had problems with both the file system and I/O subsystems in this area, where the implementation of an abstraction made certain assumptions about how it would be used that were not always readily apparent from the abstraction’s interface.

Despite its shortcomings, Linux has treated us well. Ten years ago, we would have had to license the source code to a proprietary operating system or use an in-house research operating system to implement Swarm, either of which would have limited the impact of our work. Linux has allowed us to implement Swarm in a real operating system that

is used by real people to do real work. With a little more work in the areas of documentation, development environment, and software engineering practices, Linux has the potential to be an excellent platform for systems research and development.

5 Acknowledgements

We would like to thank Tammo Spalink and Scott Baker for their help in designing and implementing Swarm. We would also like to thank our shepherd, Stephen Tweedie, for his helpful comments and suggestions. This work was supported in part by DARPA contracts DABT63-95-C-0075 and N66001-96-8518, and NSF grants CCR-9624845 and CDA-9500991.

6 Availability

For more information about Swarm, please visit <http://www.cs.arizona.edu/swarm/>.

References

- [1] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993.
- [2] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997.
- [3] John H. Hartman, Ian Murdock, and Tammo Spalink. The Swarm scalable storage system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.
- [4] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM*

Transactions on Computer Systems, 13(3):274–310, August 1995.

- [5] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [6] Norman C. Hutchinson and Larry L. Peterson. The x -kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [7] Bulter W. Lampson. Hints for computer system design. *Operating Systems Review*, 17(5):33–48, October 1983.
- [8] David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, September 1988.
- [9] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [10] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, June 1985.
- [11] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

DMFS – A Data Migration File System for NetBSD

William Studenmund
Veridian MRJ Technology Solutions
*NASA/Ames Research Center**

Abstract

I have recently developed DMFS, a Data Migration File System, for NetBSD[1]. This file system provides kernel support for the data migration system being developed by my research group at NASA/Ames. The file system utilizes an underlying file store to provide the file backing, and coordinates user and system access to the files. It stores its internal metadata in a flat file, which resides on a separate file system. This paper will first describe our data migration system to provide a context for DMFS, then it will describe DMFS. I also will describe the changes to NetBSD needed to make DMFS work. Then I will give an overview of the file archival and restoration procedures, and describe how some typical user actions are modified by DMFS. Lastly, I will present simple performance measurements which indicate that there is little performance loss due to the use of the DMFS layer.

1 Introduction

NASStore 3 is the third-generation of mass storage systems to be developed at NAS, the Numerical Aerospace Simulation facility, located at NASA/Ames Research Center. It consists of three main parts: the volume management system (volman), the virtual volume management system (vvm), and the DMFS (data migration file system) system.

A complete description of the NASStore 3 system is beyond the scope of this document. I will give a brief overview, and then describe DMFS's role in NASStore 3.

1.1 volman system

The volman system is responsible for keeping track of the tapes, and for bringing them on line when requested.

*Present address: Zembu Labs, 445 Sherman Avenue, Palo Alto CA 94306, wrstuden@zembu.com

It was designed to support the mass storage systems deployed here at NAS under the NASStore 2 system. That system supported a total of twenty StorageTek NearLine tape silos at two locations, each with up to four tape drives each. Each silo contained upwards of 5000 tapes, and had robotic pass-throughs to adjoining silos.

The volman system is designed using a client-server model, and consists of three main components: the volman master, possibly multiple volman servers, and volman clients. The volman servers connect to each tape silo, mount and unmount tapes at the direction of the volman master, and provide tape services to clients. The volman master maintains a database of known tapes and locations, and directs the tape servers to move and mount tapes to service client requests. The client component consists of a set of programs to monitor activity and tape quotas, and a programmatic interface so that programs can make requests of the volman master and servers.

1.2 vvm system

The vvm system provides a virtual volume (or virtual tape) abstraction and interface for client programs. Its main utility is to improve tape usage efficiency. Many streaming tape technologies do not handle writing small files efficiently. For instance, a seven kilobyte file might take up almost as much tape space as a one megabyte file. To better handle such cases, vvm clients (such as the DMFS system) read and write files out of and into virtual volumes (vv's), which are typically between 50 and 300 megabytes in size. These vv's are read and written to tape. Thus all tape operations involve large reads and writes which are much more efficient.

The vvm system consists of a server, client programs, and a tape server daemon. The vvm server is responsible for remembering which vv's have been allocated, and on which tape volume they reside. When a vvm client requests a vv, the vvm server contacts the volman system (the volman master specifically) to arrange for the needed tape volume to be mounted. The vvm tape

server daemon, which runs on the same machine as the volman tape servers, then reads the tapes and makes the vv available to the client. The clients obviously use the vv's to store variable-length files. There are client programs which are able to monitor the status of vv's, and there is a programmatic interface which lets programs make requests of the vvm system.

1.3 DMFS system

The final component of the NASTore system is the DMFS system. This consists of *dmfsd*, the DMFS data migration daemon, system utilities to assist file migration, and the DMFS file system (the focus of this paper). *dmfsd* is the data migration daemon, and it runs on each host supporting a DMFS file system. It is responsible for responding to requests from the DMFS file system for file restoration. The userland system utilities in the DMFS system are responsible for archiving files and for making them non-resident. There are also utilities to permit users to force the archival or restoration of specific files.

2 Description of DMFS

The DMFS file system is the part of NASTore 3 with which most end-user interaction happens. It maintains as much of a traditional UNIX user experience as possible while providing data migration services.

2.1 Layered file system

One of the main differences between DMFS and other migration file systems (such as RASHFS, the NASTore 2 file system, or HighLight[3]) is that it is a layered file system. These other file systems merge the data migration code into the file system code, while DMFS uses the layered file system formalism to place its migration code between most kernel accesses and the underlying file system.

We perceived two main advantages and one potential disadvantage with this design choice. The first advantage we perceived is that it would be much easier to create and maintain a layered file system with data migration facilities than to add those facilities to a specific file system and maintain them over the long term. For instance, the Berkeley Fast File System (FFS)[4, 5] is undergoing change with the addition of soft updates[7]. By maintaining the file migration functionality separately, we decouple DMFS from any other FFS-related devel-

opment. Secondly, we leave the choice of underlying file system to the site administrators. If the access patterns of a site are better suited to the Berkeley Log-structured File System (LFS)[4, 6], then a site can use that as the underlying file store rather than LFS. The only limitations are that the underlying file system support vnode generation numbers, and the current metadata database requires knowledge of the maximum number of vnodes at the time of initial configuration.

The one potential disadvantage is that the use of layered file systems incurs a certain amount of overhead for each file system operation. In [2], Heidemann measures layered file system overhead to be on the order of a few percent. Section 7 will describe the limited testing we have done to date. The conclusion we have reached is that any performance penalties due to layered file system technology are certainly worth the benefits in maintainability and flexibility.

2.2 Functions

The primary function of the DMFS file system layer is to provide user processes transparent access to migrated data. It determines if an operation would access non-resident portions of the underlying file. If so, it requests that the *dmfsd* daemon restore the file, and then blocks the operation until the file is sufficiently restored so that the operation may proceed. One feature present in DMFS is that when a process is blocked awaiting file restoration, it may be interrupted. As a practical matter, a user may kill (C) a blocked process.

Another main function of the DMFS layer also is to preserve the integrity of the metadata it and the userland *dmfsd* databases keep regarding files. It does so in two ways.

First, it records the generation number of the underlying vnode. When an operation (such as name lookup) causes the allocation of a DMFS vnode, the recorded generation number is compared with the generation number of the underlying vnode. In case of a discrepancy, the information in both the kernel and userland databases is invalidated. Likewise these databases are invalidated when the last reference to an unlinked vnode is released.

The second area of metadata preservation keeps a file from getting in an inconsistent state. The present metadata format permits recording that a portion of an archived file is resident, but does not permit noting that only a portion of a file is archived. The possibility of file inconsistency would arise if the on-disk portion of

a non-resident file were permitted to be modified. For simplicity, such cases are forbidden at present. Thus any writes or truncations to a non-resident file are blocked until the file is fully restored.

2.3 Metadata

The DMFS layer keeps a small amount (80 bytes) of metadata for each vnode in the underlying file system. At present these data are kept in a flat file database indexed by vnode number which is stored on a separate file system. The metadata file contains a header which records the version number of the metadata file and of the nodes, their size, and the byte order in which they are stored. The metadata include:

flags

Flags indicating state of the DMFS node, including: metadata valid, file is archived, and file is (partially) non-resident.

generation number

The generation number of the current file using this vnode number.

archive size

The size of the file (including any non-resident portion).

byte barrier

How much of a non-resident file is present on the underlying file store. Reads of non-resident files which access only data that is present on the underlying file store are not blocked.

bfid

“Binary File Identifier” a 16-byte endian-independent tag assigned to the file which is intended to uniquely identify it. The *dmfsd* databases use this tag as the main index for the file.

atime and mtime

Access and modification time stamps (with nanosecond resolution) for the file. These times are not updated due to either archive or restore agent operation.

archive time and restore time

Time stamps (with second resolution) of last archive and restore events.

For files with valid metadata, the **atime** and **mtime** values above are reported in a `VOP_GETATTR()` operation, masking archive and restoration agent activity

from view by users. When a file is flagged as non-resident, the **archive size** value is also reported in a `VOP_GETATTR()` operation. These elements together serve to preserve a more traditional user experience with files on the DMFS layer. The activities of the archive and restore agents are not noticeable by userland processes such as `ls -l`, `find -amin`, `find -mtime`, etc., other than as described in section 3.9¹.

The archive and restore time stamps are recorded to help the policy engine in the userland migration agent know more of a file's access history.

2.4 Comparison with other projects

Some of the design goals of the DMFS layer are best understood in terms of some of the data migration efforts which have come before it.

The DMFS layer is based on NASA's experience with the RASHFS file system used in the NASStore 2 system. NASStore 2 was deployed at NAS from 1991 until June, 1999. It was initially deployed on an Amdahl 5880 running UTS. Later it was deployed on two Convex 3820 computers running ConvexOS. RASHFS was a modified version of the native FFS file system implementation. The changes needed to support data migration were grafted into the FFS implementation, extra space in the inode was used to store metadata, and a few system calls were modified to call into the RASHFS implementation to ensure that a file was restored before proceeding. (A file had to be fully restored before it was executed.) This implementation was very successful. The system deployed at NAS had about 4 TB of disk in the file systems and a couple of petabytes of on-line tape.

One of the difficulties with the RASHFS system is that it basically was RASH-FFS. To add the same migration abilities on top of a different file system type would require a substantial amount of re-implementation. As there is less unused space in FFS inodes in modern (4.4BSD and later) FFS implementations, it is possible that the metadata will no longer fit. This is one of the main reasons we went with a layered file system when designing DMFS. While untested, DMFS should work with NetBSD's LFS, and possibly even EXT2FS file systems, about as well as with the FFS file system. Additionally, at the same time as DMFS was being devel-

¹the fact that accessing non-resident portions of a file will block until the needed tape can be mounted and read. That will depend on the tape silo robotics, and, for tapes in off line storage, a human finding and mounting the needed tape. This delay, especially for off line storage, can be on the order of tens of minutes.

oped, there was a tremendous amount of file system development going on within the *BSD community. Kirk McKusick was working on adding "soft updates" to FFS, a technology which works to give FFS the resiliency of a journaled file system without having to have a journal. Konrad Schroder has been working to make NetBSD's LFS implementation quite robust and functional. By not integrating DMFS into a file system, DMFS stays independent of the above technology progressions – a site could choose to follow them or not, using the same DMFS code either way.

NASore 3 is not the first tertiary storage system built on top of 4.4BSD. One predecessor was HighLight, a migration file system built above LFS. The major difference between it and DMFS is that HighLight is like RASHFS in that it represents melding data migration into a file system in a very intimate manner. However, HighLight uses certain aspects of the LFS structure to its advantage. LFS operates in terms of written segments[4, 6]. A file is described as having contents in specific segments. HighLight implements data migration by including in the file system block space (where these segments may reside) the tapes in the robotics assigned to the file system. Thus when a file is migrated to tape storage, the inode is modified to point to the tape block. When a migrated file is read, portions of the on-tape segments are read onto disk. Otherwise, the file system behaves much as a non-migrating LFS. One advantage of this methodology is that it is possible to create dense files larger than the disk storage allocated to the migration file system (something not possible with DMFS).

The DMFS layer does have a few advantages over HighLight. One is that it has no knowledge of where migrated files are stored. As mentioned above, HighLight marks the blocks stored on the tape as part of the file system. Thus it requires knowledge of how many tapes of which size (in which robots) are available at file system creation time. While it certainly would be feasible to extend this knowledge base (add new tape robotics for instance) for an operational file system, keeping this knowledge separate from the migration file system is simpler. Additionally, by decoupling the tape storage information, we permit more sophisticated tape policies. For instance, NASore 3 will by default archive a file into two independent vv's (which will reside on two separate tapes) before making it non-resident (deallocating its blocks on disk). This behavior permits a level of redundancy not readily obtained with HighLight.

3 NetBSD vfs changes

A number of changes were required to the NetBSD vnode and vfs interfaces in order to get DMFS to work. All of these changes were designed to be applicable to all file systems, not just DMFS.

3.1 The OVERLAY file system

One change was the addition of a new file system, the OVERLAY file system. The OVERLAY file system is functionally similar to the NULLFS[4] file system except that it does not export the directory hierarchy into another part of the file name space. It places itself between the file name space and the overlayed file system². It is intended for certain specific circumstances, such as that of DMFS, where the layered file system has a strong need to block access to the underlying file system. DMFS does so to prevent metadata inconsistencies. One other variant of the OVERLAY file system might be a file system which attempts to validate executables with a checksum before permitting their execution. This file system would need to block access as that is very key to its security model. Unless there is a strong need for interposition, all future layered file systems should be based on the NULLFS layer rather than the OVERLAY layer as it simplifies the fcntl(2) interactions described in section 3.8.

3.2 Layering works

The most dramatic change needed to get DMFS to work was to get layered file systems working with NetBSD. The layered file system research of Heidemann [2] describes many of the advantages of layered file systems. When we began this work, we found the layered file system implementations present in all *BSDs not to be robust enough for production work. For instance, the NetBSD 1.4 NULLFS certainly will replicate a file hierarchy in another portion of the system's file name space. However simultaneous access of this replication by multiple processes, for instance a parallel make invoked with `make -j 8`, would result in kernel locking panics. Obviously a production-quality migration system

²file system mounted on `/home`, and an OVERLAY file system subsequently mounted on `/home`. Access to files under `/home` in the file name space (`/home/user1/.profile` for example) now go to the OVERLAY file system. It in turn can pass these to the underlying file system initially mounted on `/home` as it needs to. Contrast this to the case of mounting a second leaf (non-layered) file system onto `/home`. In this case, no operations would be passed to the file system initially mounted on `/home` until the second file system was unmounted.

needed a more robust implementation.

The changes described below, especially those of sections 3.3 and 3.4, were implemented at the same time, in addition to a modification to how `null_lookup()` behaved³. As such, it is not clear which change (if any) was singularly responsible for the increase in robustness, though I suspect the change to how `null_lookup()` behaves represents most of it.

After these changes were implemented, I performed a few simple tests to ensure their effectiveness. The first one was to perform a multi-process kernel make in a directory on a NULLFS file system (`cd` into a kernel compile directory and type `make -j 8`). Before these changes, such an action would promptly panic the kernel. After these changes, it did not. Additionally, simultaneous access of multiple NULL layers and the underlying layer, such as the parallel make above combined with recursive finds in the other layers, have not been observed to panic the system. The DMFS layer we have built based on this NULLFS layer has shown no difficulties (panics or otherwise) due to simultaneous multi-user access in almost one year of operation.

It should be noted that the NULLFS and UMAP layered file systems have benefited from these changes, while the UNION file system has not. It is still not considered production quality.

3.3 Most file systems do real locking

To better support the vnode locking and chaining described in the next section, all file systems (except for UNIONFS and NFS) were changed to do vnode locking. Previously only file systems with on-disk storage actually did vnode locking, while the others merely returned success. As of NetBSD 1.5, the NFS file system is the only leaf file system which does not do real vnode locking. This defect remains as lock release and reacquisition during RPC calls to the NFS server has not been implemented. As mentioned above, the UNION file system has not been updated with these locking changes.

While it is not likely that one would want to layer a file system above many of the file systems which gained true locking (such as the KERNFS layer), this change makes it easier to implement layered file systems. Lock management is one of the keys of getting layered file sys-

³directly rather than using the bypass routine. It also checks for the case of a lookup of ".", where the returned vnode is the same as the vnode of the directory in which the lookup was performed and handles it explicitly.

tems to work. By being able to rely on all leaf vnodes doing locking, the layered locking becomes much easier. Additionally the changes to add this support were not difficult.

3.4 New vnode lock location and chaining

One change made in how vnodes are locked was to implement vnode lock chaining similar to what Heidemann described in [2]. After this change, each vnode contains a pointer to a lock structure which exports the lock structure used for this vnode. If this pointer is non-null, a layered file system will directly access this lock structure for locking and unlocking the layered vnode. If this pointer is null, the layered file system will call the underlying file system's lock and unlock routines when needed, and will maintain a private lock on its vnode. As described in the preceding section, all leaf file systems other than NFS now do locking, and thus export a lock structure. Once NFS has been fixed, the only file systems which should not export a lock structure would be layered file systems which perform fan-out (such as the UNIONfile system) as they need to perform more complicated locking.

As all file systems should be doing locking and exporting a pointer to a lock structure, I decided to add a lock structure to the vnode structure and remove it from all of the leaf file systems' private data. I was concerned about a whole stack of vnodes referring to file system-specific private data, and felt it cleaner for vnodes to refer to memory contained in vnodes. In retrospect (and having completed the change), it now seems wiser to leave the lock structure in the leaf file system's private data and just require the file system be careful about managing the memory it exports in the lock structure pointer in its vnode. This change would improve memory usage in a system with multiple layered file systems by not allocating a lock structure in the layered vnodes which would go unused.

The effect of this change is that a whole stack of vnodes will lock and unlock simultaneously. At first glance this change seems unimportant, as any locking operation can traverse a vnode stack and interact with the underlying leaf file system. The advantage is three-fold. One advantage is conceptual. By having all layers use the same lock structure, the commonality of the stack is reinforced. Secondly, layered nodes do not need to call the underlying file system if the lock structure has been exported – it may directly manipulate it itself. This lack of stack traversal becomes quite advantageous in the case of multiple layered file systems on top of each other.

The third advantage is due to the lock semantics of the lookup operation on "...". To avoid deadlock, directories are locked from parent to child, starting with the root vnode. To preserve this when looking up "...", the child directory must be unlocked, the parent locked, and the child then re-locked. When looking up "." in a layered file system, with a unified lock, the leaf file system can unlock the entire stack and re-lock it while trying to obtain the parent node. If the locks were not unified and there were separate locks in vnodes stacked above the leaf one, the leaf file system would either need to somehow unlock and re-lock those locks during the lookup of "." or to run the risk of deadlock where one process had the upper lock(s) and not the lower, while another had the lower and not the upper.

3.5 New flag returned by lookup: PDIRUNLOCK

One other change has been to plug a semantic hole in the error reporting of the lookup operation. In case of an error, the lookup operation in a leaf file system is supposed to return with the directory vnode in which it was looking locked. One potential problem with this is that it is possible that the error being returned resulted from not being able to reacquire the lock on the examined directory when looking up "...". In this scenario, the lookup routine has little choice but to return an error with the examined directory unlocked. It signals this behavior by setting the PDIRUNLOCK flag which is returned to the caller. When a layered file system is maintaining its own parallel locks (if the underlying file system did not export a lock structure), the layered file system must adjust its locks accordingly.

3.6 "layerfs" library added

Before NetBSD 1.5, most NetBSD layered file systems other than UNIONFS were based on copies of NULLFS. Typically the NULLFS files were copied and renamed, the routine names were changed (the "null_" prefix changed to reflect the new file system name), and then new features were added. This behavior represents a duplication of code. In order to reduce this duplication, there is now a library of common files, "layerfs," which provide most of the NULL layer functionality. For instance the NULL layer now consists of a mount routine and vnode interface structures. The rest of the routines are in the layerfs library. The UMAP layer now shares most all of these routines, with the only difference being that it has some customized routines (bypass, lookup, and a few others) which perform its credential mapping.

DMFS also uses this library of routines. Of the 19 vnode operations handled by DMFS, 20% consist solely of calls into this library.

3.7 File Handles usable in the context of a local filestore

When communicating with the userland daemon *dmfsd*, DMFS uses file handles to refer to specific files. It does this because on all of the occasions where it needs to communicate with *dmfsd* (for instance in a VOP_READ() attempting to access non-resident data) none of the potentially multiple paths to this file are available.

To make file handles truly useful in this manner, two changes were made to the kernel. First, three new system calls were added: *fhopen(2)*, *fhstat(2)*, and *fhstatfs(2)*. For security reasons, all three calls are restricted to the superuser. *fhopen(2)* is similar to *open(2)* except that the file must already exist, and that it is referenced via a file handle rather than a path. *fhstat(2)* and *fhstatfs(2)* are similar to *lstat(2)* and *statfs(2)* except that they take file handles rather than paths.

The other change modified the operation of file handle to vnode conversion. In 4.4BSD, the only use of file handles was with network-exported file systems, specifically by the NFS server module. For convenience, the VFS operation which did file handle to vnode conversion also did foreign host export credential verification. Obviously that combination is not appropriate for a tertiary storage system. So I changed the VFS_FHTOVP() operation to just do the file handle to vnode conversion, and added a separate VFS_CHECKEXP() operation to handle the export verification.

3.8 VOP_FCNTL added

One feature which was needed by *dmfsd* and the other utilities was an ability to perform arbitrary operations, such as start archive, finish restore, etc., on the file(s) which it was manipulating. At the same time as we were addressing this need, there was a desire to add the ability to add the capability to perform file/inode operations. One such example is adding the ability to manipulate access control lists on file systems which support them.

These operations would be similar at the VFS level to *ioctl(2)s* except that they would always reach the file system, rather than possibly being dispatched to device drivers as is the case for *ioctl(2)s*.

```

Arw-r--r-- 1 wrstuden mss 2202790 Sep 14 15:14 Inside_AT.pdf
arw-r--r-- 1 wrstuden mss    7537 Nov  9 17:27 dmfs.h
-rw-r--r-- 1 wrstuden mss   36854 Dec  6 15:28 ktrace.out

```

Figure 1: Sample directory listing

While there was no objection to adding this extension to the VFS interface, the form of its programmatic interface generated controversy and much discussion on NetBSD's kernel technology email list. The desired interface would consist of a file descriptor, a command code, and a data argument (`void *`). There were three options: add a new system call with this parameter signature, overload the `fcntl(2)` interface, or overload the `ioctl(2)` interface.

In the end, I opted for using the `fcntl(2)` system call. The main reasoning is that we should allocate the entire command space reserved for the new vnode operation now – it would not be advisable to reserve some now and then add more later. There are far fewer `fcntl` operations than `ioctl` operations in NetBSD, and they are less frequently added, so it is less likely that using `fcntl(2)` as the programmatic interface and reserving a sizable command space now will impede future kernel development. Additionally, in the case of overlay-type layered file systems, different layered file systems will need to choose unique codes, and pass codes they do not understand to underlying file systems, so that tools designed to operate on one particular file system type will operate regardless of the depth of layered file systems. This requirement is easier to satisfy with a spacious reservation of command space.

The `fcntl(2)` system call takes an integer as its command code. If the most significant bit is set, the operation is now considered a request for a file system-specific operation – a `VOP_FCNTL()` call. This division leaves approximately 2^{31} commands available for traditional `fcntl`-type operations. The remaining 31 bits encode whether a value or a structure are read into or out of the kernel (3 bits), the size of data so transferred (12 bits), and the actual command code (16 bits). Half of the command space (32768 commands) is reserved for NetBSD use, while the remaining space is for a file system's private use.

3.9 New `ls -l` reporting

One user feature added to support the data migration system is a set of additional flags which indicate archive state and which are displayed via the `ls -l` command. Files typically have a “-” in the left-most column. With

this change, an archived file has an “a”, and an archived, non-resident file is indicated with an “A”. This change was implemented by increasing the functionality of the `strmode(3)` subroutine, and by adding two extra flags to the mode value returned by a `stat(2)` call. For instance in the directory listing shown in Figure 1, it is clear that the file `dmfs.h` has been archived, and the file `Inside_AT.pdf` has both been archived and also been made non-resident.

These flags are available for use by all file systems which possess the concept of file archival attributes. For instance, the NetBSD FAT (MS-DOS) file system implementation has been extended to indicate archival state using this mechanism.

4 Archival

There are two ways for an archival to be initiated. One is for a user to request a file be archived (and possibly made non-resident) using the `forcearc(8)` utility. Another would be for an archival scan process to determine that the file should be archived. After opening the file, the first step is for the archiving process to perform the `DMFS_SETAIP fcntl(2)` operation, which sets the internal `DM_AIP` (archive in progress) flag. This operation will succeed if the process has root privileges and if no other process is in the process of archiving the file. The process ID for the file is noted for reference.

The second step of the archival process is for the archiver to add the file to the `dmfsd` databases, and to copy the file to tape storage. Adding the file to the `dmfsd` databases might also involve initializing some of the metadata fields such as the `bfid`, the unique identifier for this file.

The third step is for the archiving process to set the file's `DM_ARCH` flag, which indicates that the file has been archived. The `DMFS_SETARCH fcntl(2)` operation sets this flag.

Finally, if the file is to be made non-resident, the archiver performs the `DMFS_SETNONR fcntl(2)`, which takes the amount of initial data which are to remain on the underlying file system. The DMFS layer determines the size of the underlying file, and, assuming it is greater

than the requested size, it notes the underlying file size in the **archive size** field, and truncates it to the desired length. The **DM_NONR** flag is set for the file which indicates that it is not totally resident. From this point, the file size reported will be that which was noted during this operation rather than the actual size of the remaining portion on disk.

NASore 2 truncated all files to zero residency. In NASore 3 we have added the ability to retain a portion of the file on disk. From our analysis of some new storage architectures, such as MCAT/SRB, files will actually be managed by storage agents rather than directly by the generating program. Often such systems will add a header of metadata to the beginning of the file which serves to describe the whole file. By leaving this portion on disk, we permit whatever agent is managing the storage of these data to examine the file's header without triggering a restore event. As the exact amount of storage to leave on disk is a dependent on site and storage broker configuration, we permit the archiving process to determine how much of the file should remain.

5 Restoration

Like archival, restoration can also be triggered in one of two ways. The most common method is for a user's access of a file to trigger the restore, which is performed by the *dmfsd* daemon. Another method is the use of the *frestore(8)* program, which performs the restoration directly.

In either case, the restore agent opens the file, using either the *fopen(2)* or *open(2)* system calls, with the **O_ALT_IO** flag set. This flag requires root privileges, and will permit the process to access the underlying file directly, rather than being blocked when accessing a non-resident portion.

The next step is for the restoration process to set the **DM_RIP** (Restore In Progress) flag using the **DMFS_SETRIP** *fcntl(2)*. As with the **DMFS_SETAIP** *fcntl(2)*, this will only succeed if no other process is restoring the file.

Then the restore agent reads in the relevant virtual volumes, and writes the non-resident portions of the file to disk. As the file descriptor was opened with the **O_ALT_IO** flag set, the *write(2)* calls will not be blocked and will restore the file on the underlying file store.

At various points through the restore, the restore agent may perform a **DMFS_SETBBOUND** *fcntl(2)*. This operation takes an *offset* argument and adjusts the **byte barrier** to this new value. Typically, as the file is restored, this operation is used to move the **byte barrier**, permitting blocked reads to complete as the file is restored.

The final step for restoration is for the restore agent to perform the **DMFS_FINISHARC** *fcntl(2)* and then close the file. This *fcntl(2)* call will set a flag such that when the file descriptor is closed, the file is marked fully resident and all processes waiting for the restoration to complete will be unblocked.

Finishing the restore is done as a two-step process to better support executing non-resident files. Any process attempting to execute the file will be blocked until after the close operation of the restore agent is completed, ensuring that the file will not be noted as being opened for write while attempting to execute it.

6 Behavior of typical operations

The above sections have described how the DMFS support processes interacted with the DMFS layer. This section describes how typical user process operations interact with the DMFS layer.

6.1 read(2)

The *read(2)* operation is intercepted so that an attempt to read a non-resident portion of a file is blocked until either the data are restored, or the restoration fails. A failed restoration returns an **EIO** error to the calling process.

The behavior of the restoration-checking routine is fairly simple. If no restoration is in progress, a message is sent to the *dmfsd* daemon requesting a restore. Then, if the operation was initiated by the NFS server subsystem⁴, the **EAGAIN** error is returned. For an NFSv3 mount, the NFS subsystem will return the **NFSERR_TRYLATER** error code⁵.

Then the routine enters a loop. It flags that it is waiting,

⁴detected by the presence of the **IO_KNONBLOCK** flag to the read, a flag set only by the NFS subsystem.

⁵this error code, its NFS client does not. Thus NFS exporting a DMFS layer to NetBSD clients will result in access to non-resident files returning errors to the application. Other NFS clients, such as the Solaris NFS client, will block the access and periodically retry it.

	Without DMFS	With DMFS
Creating with 10,000 64k writes	72 ± 1	71 ± 1.4
Overwriting with 1000 1024k writes	116	116
Overwriting with 100,000 8k writes	88.4 ± 1.7	88.75 ± 0.5

Table 1: Average operation times with standard deviations (seconds)

unlocks the vnode, and sleeps. The sleep is interruptible, and if it is interrupted the system call will be retried. This behavior permits users to abort a process waiting for a restoration.

Upon re-awaking, the vnode is re-locked and the file state is re-examined. If the restoration failed, EIO is returned. If the blocked read can now progress, it is passed down to the underlying layer.

For all read operations on nodes with valid DMFS metadata, the completion of the read operation is noted in the **atime** metadata field.

6.2 write(2)

The write(2) operation is intercepted in much the same manner as the read operation, and it calls the same restoration-checking routine. The two main differences are that the write operation is blocked until the file is completely restored, and that the completion of a write operation on an archived file triggers a message to the *dmfsd* daemon that the file archive has been invalidated.

As mentioned above, the write operation is blocked until the file is completely restored because the metadata format keeps archived state for the whole file, not subsections of it. Were a restoration to fail after a write modified part of an archived file, portions of the file would still be archived and non-resident (needing to be restored from tape), and other portions would be unarchived (and thus must not be restored from tape). This limitation was not considered significant for NAStore 3's target applications.

6.3 truncate(2)

Like writes, truncate operations invalidate the archived copy of the file and are intercepted, and may be blocked while file data are being restored. If the file is shrinking so that all of the new size is presently resident, the truncate operation proceeds. Otherwise a file restoration is triggered. Unfortunately, at present there is no way to request that only a portion of a file be restored, so the

whole file is restored and then truncated.

7 Performance

One drawback to the layered file design model is that it adds a certain amount of overhead to each file operation, even if the operation is merely bypassed to the underlying layer. We have performed only simple performance testing, but we have found little to no noticeable performance degradation due to the DMFS layer.

One operation which will be potentially slower on the DMFS layer is in-kernel vnode allocation, such as is the result of a lookup operation. The allocation of the in-kernel DMFS node requires reading the metadata for this node from the metadata file.

We have performed extensive usage testing of our DMFS layer. We have deployed two internal beta-test systems using two different metadata storage formats (the one described here and a predecessor which stored metadata in a large-inode FFS). In this testing, which included generating over two million files on a 155 GB file system, none of the users complained that the DMFS layer felt slow. Obviously an attempt to access a non-resident file caused a delay, but that was due to the mechanical delay of the robotics retrieving a tape.

We have performed a limited amount of quantitative testing to compare the performance of reading and writing a fully-resident file both with and without the DMFS layer mounted. I performed three tests, all using *dd* to transfer a file either from */dev/zero* or to */dev/null*. The difference between the tests was the block size used for the operations.

All tests were performed on an IDE disk in an x86-based computer running a modified version of NetBSD 1.4. The file system was built using the default parameters of 8k blocks and 1k fragments. One test was to write a 640 MB file using a write size of 64k. I observed a strong disparity of new-file creation performance over time. Initially file creation took 72 seconds, while later creations took 82 seconds. As a comparable decrease

was observed for creation with and without the DMFS layer, I attribute this degradation to disk and file system performance. The exact origin is not important, but this observation motivated all further testing to consist of overwriting an existing file.

I timed three creations without the DMFS layer, and two with. The average times are shown on the first line of Table 1. I do not believe that the presence of the DMFS layer actually improved the FFS performance, but that the variability of times reflects the simplicity of the tests. However the tests indicate that with 64k writes, the extra overhead of the DMFS layer was not noticeable and that I/O scheduling and device performance will add an amount of variability to the tests.

As shown on line two of Table 1, creating a 1000 MB file with 1024k byte writes took the same amount of time both with and without the DMFS layer. This example is similar to the previous one in that the extra overhead of the DMFS layer was not noticeable.

Both of the above tests used large write sizes to maximize performance. As such they minimized the number of times the DMFS layer was called and thus the impact of its additional computations. To better measure the call overhead, I also tried writing with a smaller block size. Line three of Table 1 shows the average times I observed when using 8k writes. Here too, no statistically significant difference was observed.

As I am using layered file system technology, I expect a certain amount of overhead when accessing fully resident files. The rule of thumb estimate I am familiar with is that this overhead should be on the order of one to two percent. My simple tests measured less, and I believe that the rule of thumb one to two percent is a good upper bound.

8 Conclusion

In this paper I have described DMFS, a layered file system developed for NetBSD to support a tertiary storage system. I have briefly described NASTore 3, the storage system of which it is a part, and described DMFS in more detail. I have described the changes to NetBSD needed to support this work. I have given an overview of how a system process interacts with the DMFS layer to archive and restore files, and I have touched on how some typical operations are affected by the DMFS layer. Finally, I presented some simple performance measurements which indicate that while DMFS might impose

a performance degradation, it is not significant and that the rule of thumb value of 1 to 2 percent is probably a reasonable upper bound for the performance penalty.

Acknowledgements

This project would not have succeeded without the assistance of others. I'd first like to thank Jason Thorpe for his advice and assistance with many of the kernel design issues raised in this work. My fellow NASTore developers, Tom Proett and Bill Ross, helped me devise how the kernel and NASTore programs interoperate. Harry Waddell and John Lekashman, our management team, encouraged and supported us during the entire development cycle. Finally I would like to thank Chris Demetriou for his design suggestions and especially for his review comments regarding this paper.

References

- [1] The NetBSD Project, <http://www.netbsd.org/>
- [2] J. S. Heidemann, *Stackable Design of File Systems*, Ph.D. Dissertation, University of California, Los Angeles (1995).
- [3] J. Kohl, C. Staelin, M. Stonebraker, "HighLight: Using a Log-structured File System for Tertiary Storage Management", Proceedings of the San Diego Usenix Conference, January 1993.
- [4] M. McKusick, K. Bostic, M. Karels, J. Quarterman, *The Design and Implementation of the 4.4 BSD Operating system*, Addison-Wesley Publishing Company (1996).
- [5] M. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3, pp 181-197, August 1984.
- [6] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "An Implementation of a Log-Structured File System for UNIX", Proceedings of the San Diego Usenix Conference, pp 201-218, January 1993.
- [7] M. McKusick, G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem", Proceedings of the Freenix Track at the 1999 Usenix Annual Technical Conference, pp 1-17, January 1999.

A 3-tier RAID Storage System with RAID1, RAID5 and compressed RAID5 for Linux*

K. Gopinath Nitin Muppalaneni N. Suresh Kumar Pankaj Risbood

*Computer Science and Automation Dept.
Indian Institute of Science, Bangalore*

`gopi@csa.iisc.ernet.in`

Abstract

This paper presents the design and implementation of a host-based driver (a “volume manager”) for a 3-tier RAID storage system, currently with 3 tiers: a small RAID1 tier and larger RAID5 and compressed RAID5 (cRAID5) tiers. Based on access patterns (“temperature”), the driver automatically migrates frequently accessed data to RAID1 while demoting not so frequently accessed data to RAID5/cRAID5. The prototype system, called “Temperature Sensitive Storage” (TSS), provides reliable persistence semantics for data migration between the tiers using ordered updates or logging. Mechanisms are separated from policies through an API so that any desired policy can be implemented in trusted user processes. We also discuss the problems faced while moving from the original implementation on the Solaris platform to Linux. Finally, we present comparison of the performance of our design with comparable systems using striping or RAID5.

1 Introduction

The need for reliable, efficient, fast and easily manageable storage has dramatically increased because of the Web as web servers and database servers need to have these properties. The management cost nowadays is much higher than the actual storage cost, often by a factor of 4 to 7. Consider the case of a caching web proxy that maintains a large cache in persistent storage. But all the cached data is not useful; hit rates have been reported only in range of 30%. The performance of the caching proxies and database servers can be improved if it is possible to have storage on a device which allows fast retrieval of the frequently accessed data. At the same time, the storage should be reliable, i.e. it should be able to sustain disk failures without bringing the system down, a necessity for highly available applications. Further, even if the system crashes, it should be able to recover from the crash as soon as possible so that system down time is minimal. Another desirable feature is the efficient use of available storage, due to the huge

amount of data that database and web servers may potentially handle.

Disk access patterns display good locality of reference [RW93], especially in non-scientific environments. For achieving cost-effective storage systems with terabytes of data, such locality can be exploited by using a multi-tiered storage system with different price-performance tiers that adapts to the access patterns by automatically migrating the data between the tiers.

This approach has a lot in common with memory caches. Like caches, we try to improve the performance using a small faster storage layered on top of a bigger, slower and relatively cheaper storage. But our problem is different from caches in that the caches need not provide reliable persistence semantics in the event of a system failure. Next, the latencies in our case can be much longer due to the use of lower speed devices such as disks when compared with memory caches. Also, in caches, the same data can occupy space in multiple tiers; in our case, data can be resident in only one tier. Our approach also has some similarities with hierarchical storage management (HSM) solutions. However, HSM is a more general storage system comprising of secondary (disks) and tertiary storage (tape), while our solution uses only secondary storage.

Our design currently has 3 tiers: declustered RAID1, RAID5 and compressed RAID5 (cRAID5). Redundant Arrays of Inexpensive Disks (RAID) is a technique to improve the reliability and performance of secondary storage. Of the various levels of RAID discussed in [CLG⁺94], RAID1 and RAID5 have become more popular due to ease of use and price/performance respectively. Mirroring or RAID1 maintains two copies of the same data and generally provides best performance and is easier to configure. Rotating parity scheme or RAID5 costs the least of all the RAID levels for the reliability and performance it provides. It suffers from poor small update performance and configuring RAID5 is more involved. Declustered RAID1 differs from RAID1 in that the data is striped across multiple disks. Two physical stripes constitute a RAID1 logical stripe with each stripe unit data being present in *two* different disks, thus

*This work has been supported by Veritas Software, Pune/MtView but they are not responsible for its contents. Nitin is currently at Veritas Software, Pune; Suresh at Lucent Technologies, Bangalore and Pankaj at Lucent Bell Labs, NJ

ensuring resilience to a single disk failure. cRAID5 is the same as RAID5 except that the data is compressed before writing. Parity is computed on the written (compressed) data; typically, the parity computation has to be done on the compressed data of more than one stripe. Random read access into a cRAID5 stripe is handled by decompressing just the compressed stripe (and not the full stripe as we have locking at the right granularity). However, writes are more involved which we discuss later.

1.1 Design Approaches for Multi-tier Storage Systems

A multi-tiered storage system can be implemented at various levels:

As an application This is the most flexible approach as the user application has complete control over data management. Applications like database servers are good candidates. In a file system environment that supports Data Management API (DMAPI) [DMAPI], an application can keep track of access patterns of files and implement a multi-tier storage. HSM implementations using DMAPI already exist.

Inside the file system Less flexible than the former, but may provide better performance than the former due to lower context switch overhead.

As a layered device driver (our approach) The device driver is layered on top of block storage media (generally disks). A given I/O request is divided into separate I/O requests each of which is issued to the device drivers of the underlying storage media it is configured to use. Our design currently has 3 tiers: declustered RAID1, RAID5 and compressed RAID5 (cRAID5), with future extensions for additional tiers like NVRAM. The data is migrated between the tiers depending on access patterns, so as to keep frequently accessed (hot) data in RAID1 tier and not so frequently accessed (cold) data in RAID5/cRAID5.

In the controller HP AutoRAID is a two-tier RAID storage array that uses RAID1 and RAID5. It is implemented at the controller level communicating with the host over the system bus. SCSI disks are connected to this controller through the controller's internal SCSI bus.

1.1.1 Motivation for a Layered Driver Approach

An application based approach would only improve that particular application while a file system based approach does not help configurations that use a block device directly such as a database configured to use raw devices.

There are both benefits and costs in either the controller (often called hardware RAID) or device driver (software RAID) approach. We briefly discuss them here (for a more detailed comparison, see [Veritas]):

Benefits of a Controller Based Approach

Lower I/O bus traffic As the host issues a logical I/O to the controller, the I/O bus will see much less traffic com-

pared to a software approach. For instance, a mirrored configuration of software RAID will issue two separate I/Os, one for each copy.

Separate Hardware A controller approach uses a separate on-board processor to process I/O requests, thus not loading the host processor. In parity based configurations like RAID5, this can save the host some computing, but with very fast server processors nowadays, this is a minor issue. In addition, controllers may have on-board NVRAM to improve write performance. But all these add to the cost of the system.

Benefits of Device Driver Approach

Host processors easier to upgrade The problem of increased I/O bus traffic and load on the host processor need not always be that serious. Host processors are generally far more faster and are easier to upgrade than the ones used in controllers.

Controller level redundancy In controller based approaches, redundancy is provided against disk failures but providing redundancy for other hardware such as controller hardware, can increase the cost. In software based configurations this can easily be provided by connecting disks across different controllers. Also, this can lead to better performance as the load is distributed across multiple controllers.

Flexibility A controller based approach has limitations on the number of disks that it can manage. Once that limit is reached, any new disks have to go under another controller and so cannot be accessed by the former. Also, configuration flexibility is limited. A software approach can be easily customized for a new access pattern. As we shall show in the section on implementation, the mechanisms and policies can be cleanly separated in a software approach.

A device driver approach is much simpler from an infrastructural viewpoint and it is increasingly becoming important commercially as many vendors are now providing such solutions for managing multiple disks (Veritas VxVM, IBM & HP LVM, SGI XLV). Hence we have investigated the device driver approach. Implementing at the controller level was not considered seriously due to the difficulty of carrying out modifications at this level, lack of information, and also the time and cost involved in developing the skills and the cost of implementation in a university laboratory environment.

In section 2 we discuss related work. Section 3 explains the design. Section 4 covers the implementation, with specific details on the changes for Linux from our initial Solaris prototype. Performance study for the Solaris platform is presented in section 5. Section 6 draws conclusions and spells out further work.

2 Related Work

Loge [ES92] and Mime [CEJ⁺92] disk controllers use a level of indirection to adaptively alter the physical location of data to improve performance. [AS95] present a device driver implementation of a related idea.

HP AutoRAID [WGSS95] is a firmware implementation of the idea. It is implemented at the controller level, communicating with the host over the SCSI bus. It has a separate on-board processor to do operations such as parity computation, maintaining various data structures, tracking access patterns and effecting migrations between the RAID1 tier and the RAID5 tier. It uses on-board NVRAM to improve writes and does log-structured updates to RAID5. It maintains logical to physical translation tables and two other tables for RAID1 and RAID5. The logical to physical translation makes the migrations of data between RAID1 and RAID5 transparent to the user.

[MK96] presents orthogonal placement of data to improve the performance of RAID5 in both normal and degraded mode.

[HD89] discusses chained declustered RAID1. Declustered RAID1 differs from RAID1 in that the data is striped across the disks and two physical stripes constitute a RAID1 logical stripe. Our driver's RAID1 tier is implemented as a more flexible version of declustered RAID1. Since our experimental setup consists of a single controller, we observed poorer performance for declustered RAID1 than RAID1. On a multi-controller configuration, it might provide better performance than RAID1.

[KL96] explains how compression algorithms can be used to predict future accesses with high probability by the use of access patterns and perform prefetching. But there is a sizable memory requirement for maintaining the required data structures, and an in-kernel implementation would pin down most of main memory. We show how it can be implemented in a user process¹ that uses the application interface provided by our driver to keep track of accesses and effect migrations.

Linux has an implementation for RAID personalities including RAID0, RAID1, RAID4, and RAID5 as *md* (multiple disk device driver). It can be called a device driver as it occupies a major number, but it actually never services any I/O requests. All the I/O requests are mapped to the respective underlying device driver even before they reach the strategy routine of *md*. The implementation is a kind of hack in the kernel (necessitating changes in the kernel code only for *md*), and thus does not follow the framework of a standard Linux device driver. As explained below, this has been necessary until 2.1 kernels as it was not possible to use concurrency amongst multiple devices managed by a single device driver.

3 Design of the Storage System

3.1 Design Principles

1. Use only commodity hardware that is available in typical systems.
2. Avoid storage media dependencies such as use of only SCSI or only IDE disks.
3. Keep the data structures and mechanisms simple.
4. Support reliable persistence semantics.
5. Separate mechanisms and policies with the former inside the driver and the latter in user level applications.

Our driver uses the host processor for performing operations like I/O processing and parity computation. As the driver runs in kernel mode with forced context switching turned off, increased load on the host processor can slow down the system, and result in poorer system response. This forced us to keep the data structures and mechanisms simple. We believe this tradeoff to be advantageous overall.

Our design does not assume any special hardware such as NVRAM or dedicated processor which are generally not available in typical workstation systems. But if NVRAM is available, it can be used (a ramdisk type driver is all that is required). Also we should be able to use the existing storage media; the driver can work with any devices with a block device driver. This keeps the design flexible but makes us unable to use device specific optimizations.

To guarantee reliable persistence semantics, we have investigated making changes to the state of a stripe using both *ordered updates* and using a separate logging device. State changes occur due to migrations. The second approach is especially suitable in the presence of cRAID5 in the system as many writes become read/modify/write operations. In addition, it speeds up RAID5 partial writes as well as mirrored (declustered) writes for RAID1. Our first implementation on Solaris 2.5.1 used only ordered writes as we had only 2-tiers in that system. Our next prototype on Solaris used both while the Linux prototype will also use both.

The ordered updates guarantee correct operation in the event of a system failure as the state changes, but there is a possibility for some physical storage to go unaccounted during these changes, which can be reclaimed by using a UNIX *fsck*-like program that we call *device-check*. Unlike *fsck*, it is not essential to run this program every time the system crashes. Also this program can be run on a live system. The ordered updates are explained further in the section on migrations.

We have provided enough hooks to implement policies outside the kernel. The driver provides an interface for applications¹ to access the data and services of the driver. The

¹By the term application or user process in this paper, we mean root privileged processes, such as *fsck*. Storage devices are protected resources and are generally not directly accessible by unprivileged processes.

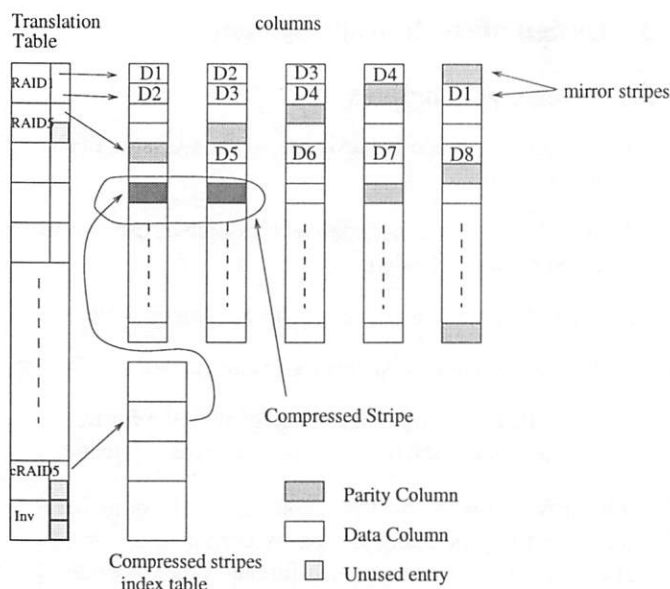


Figure 1: Storage Organization

device-check program, for instance, that is needed with the ordered write approach uses this interface.

The multi-tier storage is referred to also as temperature sensitive storage (TSS) as migrations enable frequently accessed (“hot” temperature) to reside in RAID1 and less frequently ones in RAID5/cRAID5 (“warm” and “cold”).

3.2 Data Layout

3.2.1 Physical Storage

The physical storage is organized in RAID5 fashion as shown in the figure 1. The storage consists of a set of *columns*. We use the term columns to distinguish them from disks. A column is a block device which can be a single disk or can be a pseudo device built from multiple disks using a layered driver. The smallest column limits the size of the storage. A column is divided into contiguous regions called *stripe units*. A *stripe* is a formed by grouping one stripe unit from each column.

The size of the data portion of a stripe is $N-1$ times the stripe unit size (N being the number for columns), as one stripe unit from each stripe is used for parity. We use left-symmetric parity distribution scheme[LK91]. The *polarity* of a physical stripe is defined as the column number of its parity stripe unit. The polarity is used to ensure declustering for RAID1 stripes. These physical stripes act as backing store for logical stripes that are explained next.

3.2.2 Logical Storage

The logical storage can viewed as a collection of logical stripes. The logical to physical translation is done by the driver, so that the user’s view of the data does not change

even as the stripes undergo change of state. A logical stripe can be in any of the following states:

Invalid No backing store is allocated for this type of logical stripe. Initially all the logical stripes belong to this type.

RAID1 Two physical stripes of different *polarity* provide backing store for a declustered RAID1 logical stripe. Due to the way stripe units are numbered in a stripe, this ensures that the mirrors always come from different columns. The parity stripe units of the physical stripes backing a RAID1 logical stripe are left unused. This leads to some of the storage going unused but it keeps the data structures and the mechanisms simple.

RAID5 A single physical stripe provides backing store for a RAID5 logical stripe.

cRAID5 The data of this stripe is compressed and stored in a physical stripe. Since a compressed stripe is smaller than the uncompressed one, a full physical stripe is not necessary for providing the backing store for the compressed data. The unused region of the physical stripe can be backing store for some other compressed stripes. Since the compressed data is stored in RAID5 format, we can sustain single disk failures. Since the unit of allocation is at sub stripe level, an allocation bitmap is stored persistently on the disk.

An *allocation bitmap* is used to maintain the allocation status of physical stripes. The logical to physical translation table and the allocation bitmap are persistent structures stored on private partitions² which are read into the main memory at the time of loading the driver which is a loadable kernel module. When changes to these data structures need to be persistent, like logical stripe type or allocation status of a physical stripe, the on-disk copies of those specific entries are updated.

With out compression, it is sufficient to know if a physical stripe is acting as backing store for any logical stripe or not. Introducing compression requires us to be able to address storage in smaller units than stripes. For the sake of simplicity, we set the number of individually allocatable subunits within a stripe to a power of 2 (2^n) before the creation of device; the granularity of compression is thus set to $\text{stripe size}/2^n$. Once the device is created, this cannot be changed.

If a logical stripe is backed by a cRAID5 stripe, the translation table gives the index into the compression table where the information of the backing physical stripe is stored. The latter includes the physical stripe that contains the backing store for the logical stripe, the actual size after compression and the offset of the allocation unit in the physical stripe.

²Redundancy for the information on the private partitions remains part of future work.

3.3 I/O Processing

A given logical access is divided into separate physical accesses and issued to the underlying drivers. The given access is first broken at the stripe granularity as all stripes that the access covers need not be of the same type. Each of these accesses is handled depending on the type of the stripe.

A read to an invalid type stripe always succeeds. Backing physical stripe(s) are allocated on demand for writes making the stripe RAID5, and the operation is retried.

A write to a RAID1 stripe should update both the copies, whereas a read can be satisfied with either of the copies.

Reads and full stripe writes to a RAID5 stripe are handled in normal RAID5 fashion. During partial stripe writes, an attempt is made to migrate the whole stripe to RAID1 failing which the access is handled in RAID5 fashion.

Read in cRAID5 are handled by reading the stripe units of the compressed stripe. The data is uncompressed and the requested portion is copied to the target. Currently, no migrations are done as this will be left to policy modules (see below).

For write requests, the compressed data of the stripe is read, uncompressed and the old data overwritten by the new data. The data is then compressed and written in a new stripe or possibly in the original stripe itself. The latter is the best case when the number of physical blocks allocated to the new compressed stripe does not change. Otherwise, we need to find a new stripe that can accommodate this new compressed stripe and then release the storage in the old stripe. If compression is not good enough so that we need close to a full stripe itself for the write, the above read-modify-write cycle can be dispensed with in some cases if the write is for a full stripe. However, in all the above cases, no migrations are done, leaving the decisions to the policy modules.

3.4 Migrations

Migrations result when a logical stripe changes type. A logical stripe changes from invalid to RAID5 on a write to it. After the migration is completed, the I/O is retried (this time in RAID5 fashion). If the request is a partial stripe write, another migration is triggered to make it RAID1. Thus a partial write to an invalid type stripe ultimately results in its migration to RAID1. A full stripe write to an invalid stripe only makes it a RAID5 stripe. An alternative strategy is to move the stripe directly from invalid to RAID1 on an update whether it is full stripe request or a partial one. We chose the 2-stage approach as we have no information whether the data will remain hot after this write. If it remains hot, the policy mechanisms will migrate it to RAID1. It also keeps the implementation simple and it is the right strategy for large I/Os.

RAID1 to RAID5 migration usually happens when a RAID1 stripe is victimized to give one of its physical

stripes to a currently invalid logical stripe to make it RAID5 or to a RAID5 stripe to make it RAID1.

Our strategy for migrations to/from cRAID5 is similar. First migrate the needed stripe to RAID5 (leaving the other stripes in cRAID5 with certain parts invalid) and migrate to RAID1 as needed. RAID5 to cRAID5 migrations typically take place through policy mechanisms when the data becomes cold.

We use software LRU to maintain access frequencies of stripes.

3.5 Application Interface

To keep the system flexible and the implementation simple, we have tried to offload the policy decisions to user level applications³. The driver provides interfaces using which an application (necessarily with root privileges) can access the driver data such as logical to physical translation table, stripe access information, physical stripe allocation bitmap and services such as initiating migrations, punching holes, etc.

4 Implementation

The first prototype has been implemented as a pseudo device driver on Sun Solaris 2.5.1 with two tiers of RAID1 and RAID5⁴. The next prototype added the cRAID5 layer. The current effort is reengineering it for Linux 2.2.5⁵. Since the Linux device driver interface is very different from Solaris, and given the rapidly changing internal interfaces in the Linux kernel since 2.0, this has required careful study. We will discuss some of these below (see Sec 4.2.1).

The implementation of TSS has been designed to be modular to allow each module can be tested and debugged independently. The implementation not only supports a unified, integrated TSS device but also supports RAID1, RAID5, declustered RAID1 devices also. The control and configuration of the devices is done through ioctl calls.

Each device has a *dev* structure associated with it, which contains the information about the device including its personality type, personality specific information, information of the underlying devices, and the function pointers corresponding to the entry points specific to the device personality. Figure 2 depicts the details of *dev* structure and associated data structures.

When any entry point like read, write or close is called on a device, it first executes the generic implementation as exported by the device. The generic code in turn calls the

³This is a common Unix approach, used for example, in fsck, X, to access kernel data through kmem, proc, etc

⁴The source of this is available under GNU GPL. Pl. send e-mail for the exact location.

⁵The current status of the Linux implementation: Though the prototype works correctly for the case of 5 "virtual disks" on one physical disk, it does not yet work properly for the case of 5 physical disks (the useful case).

personality specific function based on the type of the device. This design allows us to implement each personality separately and then integrate them together.

In the initialization module, a global array of pointers, to device structures is created. In the configuration ioctl, the generic device structure is created and its personality is specified, later the personality specific ioctl is used to configure the actual device parameters.

The I/O queue maintained by the driver is the same for all the devices controlled by it. When a request is in queue and `request_fn()` is called, it detaches the first request from the queue and passes this request structure to the personality specific strategy routine.

The repackaged I/O request is divided into stripe requests and all these are collected under first level stage I/O. Now the mapping from the logical to physical stripe is performed. For the declustered RAID1 and RAID5, the mappings are direct functions of the logical stripe number. For the integrated TSS device, *maptable* and *compression table* are used for this operation. Figure 3 gives the details of the way I/O staging is done at various levels in the data flow from the pseudo device to the actual device and the various abstractions involved.

We first take a look at how I/O is handled in Linux.

4.1 I/O handling in Linux

Linux uses *request* structures to pass the I/O requests to the devices. All the block devices maintain a list of *request* structures. When a buffer is to be read or written, the kernel calls `ll_rw_block()` routine and passes it an array of pointers to buffer heads. `ll_rw_block()` routine in turn calls `make_request()` routine for each buffer. `make_request()` first tries to cluster the buffer with the existing buffers in any of the *request* structures present in the device queue. A *request* structure consists of a list of buffers which are adjacent on the disk. This clustering is performed only for the drivers compiled in the kernel and not for loadable modules. If clustering is possible, no new *request* structure is created, otherwise a new *request* is taken from the global pool of structures and initialized with the buffer and is passed to the `add_request()` routine. This routine applies the elevator algorithm using insertion sort based on the minor number of the device and the block number of buffer. If the device queue is empty, the kernel calls the strategy routine i.e. the `request_fn()` of the driver; otherwise, it is the responsibility of the driver to reinvoke it from the interrupt context (see Figure 4). Another requirement for `request_fn()` is that it cannot block as it needs to be called from the interrupt context.

To allow the accumulation of requests in the device queue, a plug is used. When the request comes in and the device queue is empty, the plug is put at the head of the device queue, and a task comprising of the unplug function is registered in the disk task queue. Thus the requests keep

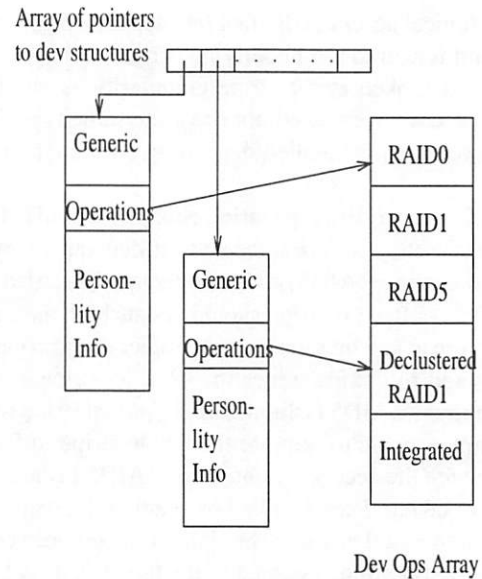


Figure 2: Device info structure

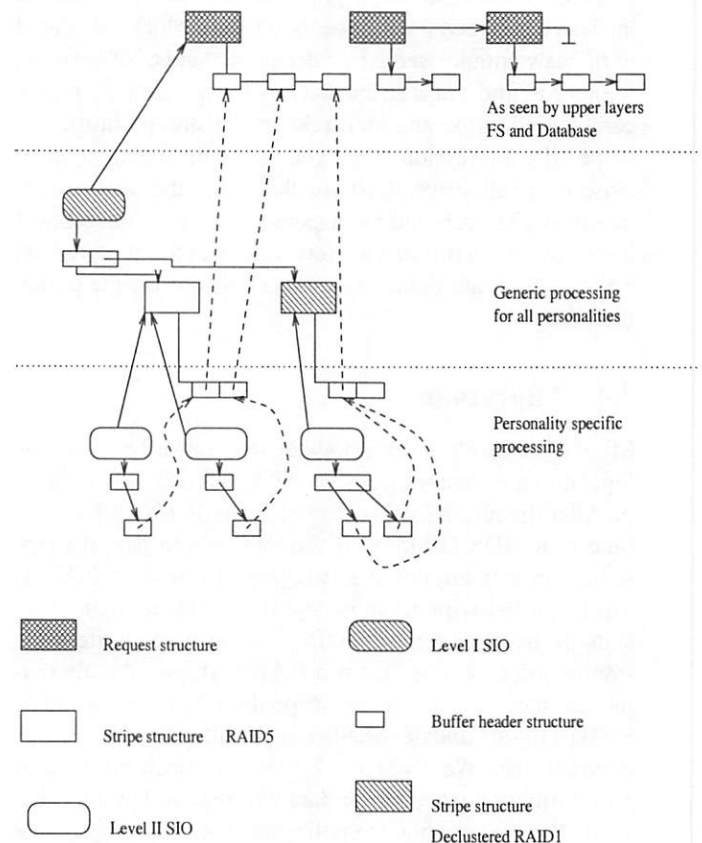


Figure 3: I/O staging Levels

on accumulating for some time and then the task queue executes the unplug routine which removes the plug and calls the `request_fn()` to service the requests.

4.2 Problems for a Linux Implementation

4.2.1 Linux Device Driver Issues

As an example, the 2.0.x versions of Linux cannot use the concurrency provided by multiple disks if its device driver framework is strictly followed. Though concurrency exists at the hardware device level, if the same driver is used for all of them, the I/O requests for different devices will be serviced sequentially. A driver can have only one queue of device requests and hence all the requests for the devices controlled by the driver will be in the same queue. The processing of a new request is initiated only when the previous I/O is finished and in interrupt context. This problem has been eliminated in 2.2.x versions by allowing drivers to register a function which returns the pointer to the head of the queue in which the new request is to be inserted. Now a driver can maintain separate queues for each hardware device.

We face the following situations in implementing a layered device driver in Linux:

Blocking in Interrupt Context For interrupt driven block drivers, the strategy routine (`request_fn`) can be called from interrupt context but it cannot block. On Solaris, this is possible as it has interrupt threads. For a layered implementation, one needs to call the `ll_rw_block` routine from the `request_fn`, so that it can put the buffers in the request queue of the underlying device.

But `ll_rw_block` routine in Linux can block as it has a global array of request structures, and if all the slots in the array are filled then the function has to block. One solution could be to modify the `ll_rw_block` code so that if we cannot find a request structure, we return immediately and queue a task in schedule queue, to be executed later.

A better solution would be to make sure that we never need to call strategy routine in the interrupt context. This can be done by consuming all the requests queued to the device queue in a single invocation of the `request_fn`. This is so as the kernel calls the `request_fn` from process context only if the device queue is empty.

The solution to this problem is to design the `request_fn()` in such a way that it keeps on executing till all the requests in the device queue are exhausted. Thus it will always execute from the process context. One drawback of this scheme is that one process may have to be delayed or blocked for I/O requested by some other process, but this is acceptable as the situation will occur only when all the request structures are exhausted which is likely to be infrequent. The pseudocode for `request_fn()` is as below:

```
tss_strategy() {
    while (1) {
```

```
        if (no request in queue) return
        remove first request from queue
        get tss dev corresp to minor#
        in request
        call personality specific strategy
        if (error in delegating I/O)
            call end_request with buffers
            not uptodate
    }
```

```
}
```

Fixed Size Buffer The buffer size for a device is fixed, unlike Solaris where we can have variable sized buffers. For example, to implement RAID5 efficiently, we need to distinguish between the full stripe write and partial stripe write as the latter involves a read-modify-write cycle. In Solaris, this is easier as one buffer can span across stripes. In Linux, each logical buffer is already split into smaller fixed sized buffers, so one has to rediscover the logical buffer to distinguish between the two cases and do the processing accordingly.

In addition, reporting of errors when they occur has to be at buffer granularity. We can keep track of errors only at the individual buffer and therefore cannot do error reporting at the stripe level.

end_request If we need to use multiple queues, then the current `end_request` does not work. We need a new implementation.

4.2.2 Lack of other support in Linux kernel

In addition to the above problems arising due to lack of infrastructure in the Linux kernel for layered device drivers, we have the following additional problems, due to lack of other required support in the Linux kernel. Firstly, there is a **cache consistency** problem that occurs in case of RAID5 writes. RAID5 writes are of two types. The full stripe writes completely bypass the buffer cache as I/O is done by creating only the buffer header structures with the data pointers appropriately set, and later releasing them. The partial stripe writes go through buffer cache as they involve a read-modify-write cycle. Thus the cache can become inconsistent. To eliminate this problem, buffers for the stripe undergoing a full stripe write need to be invalidated.

Second, Linux does not have an implementation for **condition variables** to allow atomic grabbing of a lock with condition checking. *ilock* had to be implemented to check for any overlap among the various I/Os being issued concurrently.

When a request comes in, the region that needs to be locked is calculated in terms of starting and ending sector. First, a region structure is created with this information, the global lock that protects the list is then acquired followed by disabling of the interrupts. This whole exercise ensures that any addition or deletion to or from the list is atomic. Now each region structure in the list is compared for any

overlap with the new structure. If no overlap is found, a lock structure is allocated and initialized. The mutex lock in the lock structure is acquired and the reference count is set to 1. Now the list lock is released followed by reenabling of the interrupts.

If an overlap occurs, then the reference count of the associated lock structure is atomically incremented, the list lock is then dropped. Now the process does a down on the mutex of the lock structure corresponding to the overlapping region. Linux reenables the interrupts when the process tries to sleep. By exploiting this feature, we can do checking of overlap and wait in an atomic fashion.

When the process wakes up, it atomically decrements the reference count of the lock structure on which it was waiting. If it is zero, it frees the lock structure. Now it starts all over again to check for overlaps. Following is the pseudo code:

```

create region struct for request
rep: grab the list lock
disable the interrupts
check for overlap
if(overlap occurs) {
    release the list lock
    wait on mutex & enable interrupts
    atomically decrement the refcnt
    if zero, free lock structure
    goto rep
} else {
    create a lock struct
    set its refcnt=1 & lock its mutex
    insert region struct in list
    release the list lock
    enable interrupts
}

```

When an I/O is done, the associated lock structure's reference count is decremented in the interrupt context and tested for zero. If it is zero, the associated lock structure is freed. The region structure associated with the done I/O is also removed from the list and freed. The pseudo code is given below:

```

decr refcnt of associated lock struct
if (lock refcnt is zero)
    free the lock structure
else wake up those sleeping on mutex
free the region structure in list

```

4.2.3 Other problems not unique to Linux

One important problem is that there is no API between users of a device and its driver. The size of the TSS device is not fixed but varies dynamically as the organization of data changes among different storage personalities. If there exists a file system that can talk to underlying device through an appropriate interface and can dynamically

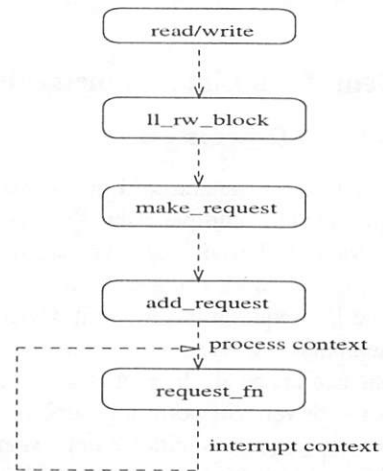


Figure 4: I/O flow in Linux

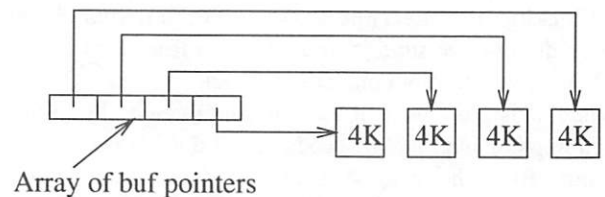


Figure 5: Data structure passed to ll_rw_block()

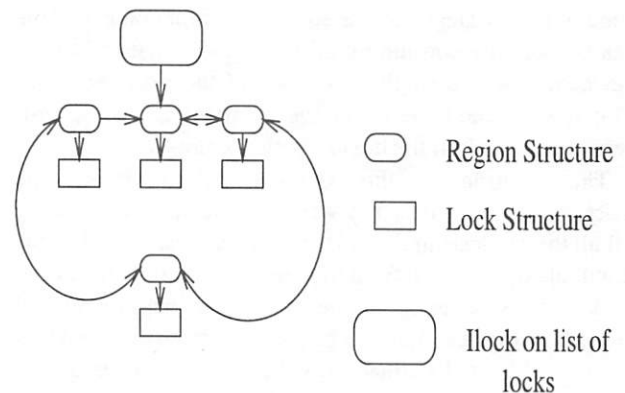


Figure 6: Interlock Structure

change the file system size, this feature of dynamic growth or shrinking feature of TSS can be exploited. None of the file systems available on Linux platform have such a facility (and most other FS on most OSes too!), so we still need to export a fixed size of the device (required at time of mount). During device configuration, the user needs to specify the fraction of device that can be under RAID1 and fraction that will be compressed. Using this information the device size is calculated and exported. Ambiguity still exists as the amount of compression achieved cannot be predicted in advance.

4.3 Changes to Kernel

Even though the device driver framework of the Linux has been strictly followed, there are a few changes that are still needed to the kernel:

Enable Clustering for TSS device Linux does clustering only for drivers compiled into the kernel and not for loadable modules. TSS relies on clustering to gain performance, otherwise all the stripe I/Os will be partial ones. So we need to enable clustering for the TSS driver in `make_request()` code.

Limiting the number of requests For each request structure that gets queued to the TSS device queue, there will be requests queued in the underlying device queue. So if all the request slots are consumed by the TSS device, it will lead to deadlock, as processing these requests require a free request structure. To avoid such deadlock, the number of request structures that can be captured by the TSS device is set to half of the total possible request structures in the system.

Removal of Plugging Plugging of the device queue is done in Linux to allow for the accumulation of the requests in the queue. For a pseudo device this accumulation doesn't make sense as this will be done again at the underlying device level. Thus `ll_rw_block()` code needs to be changed to bypass plugging for the TSS device.

4.4 Data Structures

Maptable The maptable is a data structure used for logical to physical translation, and maintaining access information. This is read into the main memory at driver loading time. Whenever an entry changes its type, it is flushed to stable storage but access information updates do not result in flushing. The entries of this data structure are 64bit long, containing fields for the type of the stripe, the physical stripes backing the stripe, the access information and an advisory bit to note if there is an access currently active on the stripe (see figure 8). The size of the maptable is equal to the number of logical stripes in the configuration.

Allocation Bitmap This is used to maintain the allocation status of physical stripes. Since the unit of allocation is smaller than the size of the stripe with cRAID5, we have a bit for each allocation unit of storage.

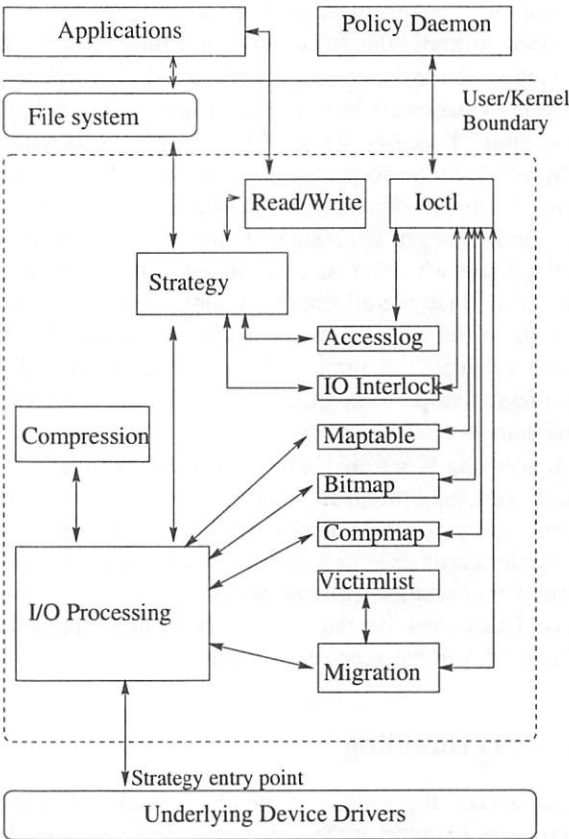


Figure 7: Implementation Model

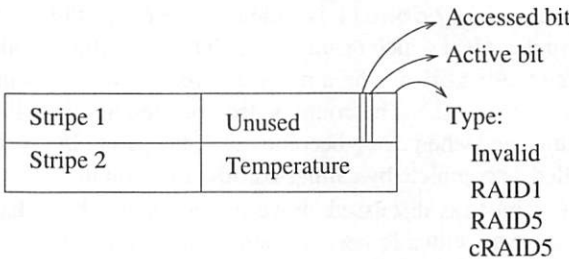


Figure 8: Maptable Entry

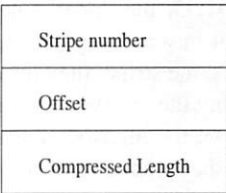


Figure 9: Compression Table Entry

Compression Table This table stores the metadata of each of the compressed stripe. For each compressed stripe, we need to know the stripe number which provides the backing store for the compressed data. Also, since the allocation of backstore is not done in units of bytes but in some units of sectors, we need to store the exact number of bytes which the compressed stripe occupies. This is required for the decompression algorithm. Also since the allocation is done in units smaller than the stripe size, we need to know which units are occupied by this compressed data. But since the allocation is contiguous, we need to store the offset of the first unit within the stripe. We can always calculate the number of units, from the size of the compressed stripe. Thus the structure of the compressed table entry is as shown in the figure 9.

Access Log When enabled, all the accesses to the device driver are logged in an in-memory access log which is a circular array of access entries. Each entry consists of two fields; the starting sector number and the size of the access in number of sectors. This can be enabled on a per instance basis. This access log can be accessed from an application to keep track of the accesses to the driver.

4.5 I/O Handling

On an access, the region spanned by it is locked. This is required as the type of the stripe should not change when an access is in progress. The access is then subdivided into separate subaccesses at the stripe granularity. These are then grouped under a *StageIO* and issued to the underlying storage drivers. The StageIO maintains a count of the number sub I/Os. In UNIX, whenever a block I/O request completes, `biodone()` is called for the request in the interrupt context which results in a call to the routine pointed to by the `biodone` field of the I/O request data structure (`struct buf`). The count is decremented in this calling routine and when count becomes zero, the parent I/O is signalled as complete by calling `biodone()` on it.

In Linux, as discussed above in Section 4.2.1, we have to come up with a few workarounds for a similar effect.

I/O Interlocking is not critical for non redundant storage but for redundant storage schemes like RAID it is essential for correct operation. For RAID1, we need to interlock the I/O region as two concurrent updates can leave the two copies (the data and the mirror) out of sync.

For RAID5/cRAID5, the updates need not be overlapping either to result in wrong operation. If two update accesses come to the same stripe, they need to be serialized as both will be updating the parity (in addition to the data). In addition, in cRAID5, the interlock has to be set at the logical stripe level so that atomicity of the write is maintained in case of simultaneous accesses to the compressed stripe.

In addition, we need to make sure that the type of a logical stripe does not change when an I/O is in progress on it.

4.6 Further Details of Linux I/O Processing

Processing of the stripe level I/O now depends on the type of the stripe, as explained below.

4.6.1 Declustered RAID1 I/O Handling

For a write request, corresponding to each buffer in the stripe, two buffer heads are created. For a read request only one buffer head per buffer is needed. These buffers are grouped under a second level stage I/O and are issued to the underlying devices to which the logical buffer maps.

4.6.2 RAID5 I/O Handling

I/O handling depends on whether it is a partial stripe I/O or full stripe I/O, so we discuss them separately.

Full stripe I/O Parity is computed from the data, and then data and parity both are written by breaking the stripe I/O into second level stage I/Os corresponding to each buffer in the stripe. We also need to invalidate the cache buffers corresponding the stripe undergoing I/O before writing the data to disk (see section 4.2.2).

Partial stripe I/O Partial stripe writes are a bit more involved than full stripe writes. To improve performance, a partial stripe write is handled in two ways:

1: *Read-Modify-Write*: This type of I/O handling is used when the size of update is less than half the stripe length. The buffers in stripe units that are going to be overwritten are read, then the parity is computed by xoring the old data, old parity and new data. The new parity and data are written to the disk by creating second level stage I/Os for each of the buffer involved.

2: *Reconstruction-Writes*: This type of I/O handling is used when the size of update is more than half the stripe length. Stripe units that are not going to be written are read in from the disk. The parity is computed by xoring the data read and data to be written. The new parity and data are written in similar way to Read-Modify-Write I/O.

4.7 Implementation of Migration

Here we discuss only the ordered write strategy for migrations between RAID1 and RAID5 for ensuring reliable persistence semantics⁶. We are currently investigating policies for migrations to/from cRAID5, mostly in a prefetching/caching framework[CFKL95, KTPB96].

In ordered write strategy, the updates are performed in a well defined order. We now explain this ordering of events and show how it does not result in wrong operation in the event of a crash.

Invalid to RAID5

⁶We omit discussion of the logging approach as it has been currently implemented on Solaris only.

1: The allocation bitmap is checked for a free physical stripe. If a free physical stripe is found, it is marked allocated and the entry of the bitmap is flushed to stable storage. The physical stripe is returned to the caller.

2: If no free physical stripes are left, a RAID1 stripe is victimized migrating it from RAID1 to RAID5 (this conversion itself involves multiple steps which are explained next), and one physical stripe is returned to the caller.

3: The maptable entry of the stripe is marked RAID5 and flushed to stable storage. The access is retried.

If the system fails between steps 1 and 3 or between 2 and 3, some storage may go unaccounted but does not result in any incorrect operation. This storage can be recovered online using the *device-check* program (see the section on application interface).

RAID1 to RAID5

1: An interlock is set on the stripe so that no one can access this stripe.

2: Full stripe data is read in, the parity is computed and the parity stripe unit of the physical stripe to be retained is updated.

3: The maptable entry of the stripe is marked RAID5 and flushed to stable storage. The physical stripe currently unused is returned to the caller.

Steps 1 and 2 do not change the structural state information of the system in any persistent way. If the system fails just after completing step 3 (marked RAID5, metadata flushed), one physical stripe goes unaccounted but cannot result in incorrect operation. The lost stripe can be recovered by running *device-check*.

RAID5 to RAID1

In the current implementation, a partial stripe write access to a RAID5 stripe triggers this migration. A more general scheme might wait till the stripe is accessed frequently enough and then attempt migration. To make transition from RAID5 to RAID1, a stripe requires a physical stripe whose polarity is different from the physical stripe currently used.

1: The allocation bitmap is searched for a free physical stripe whose polarity differs from the currently used physical stripe.

2: If the required physical stripe cannot be found in the free list, a RAID1 stripe is identified and victimized, migrating it to RAID5.

3: The full stripe data is read in and the physical stripe obtained from the victimization is updated with this data.

4: The maptable entry of the stripe is marked RAID1 and flushed to stable storage.

Steps 1 and 2 change the structural state information, but only one of them do it as step 2 is attempted only when step 1 fails. Step 3 does not change the structural information in any persistent way. If the system crashes before step 4, one physical stripe goes unaccounted which can be recovered by using *device-check*.

4.8 Optimizations to Enhance Performance

4.8.1 Victim list Management

The best candidate for victimization is a RAID1 stripe that has not been accessed for the longest time. But we found searching through the entire maptable for finding such a *perfect* victim to be extremely time consuming loading the host CPU. The problem is even more serious for a kernel module as kernel threads are not preempted when they run out of their time slice (even in Solaris 2.5.1, a kernel thread is preempted only if a higher priority thread becomes runnable), affecting all other processes and drastically reducing the system response. To reduce the amount of searching, we implemented *victim lists*.

A victim list is a list of RAID1 stripes that are sorted in the increasing order of access frequency. This list is consulted to find a victim but only as a hint. The stripe is again checked to see if it still remains a RAID1 stripe, or if it had any accesses after the (re)construction of the list, or if there is an access currently active against it. If any of these conditions is true, the entry is dropped and the next entry in the list is checked.

When the list is exhausted, it is reconstructed by searching through the maptable populating the list with RAID1 stripes that do not have their access bit set, and do not have any accesses active against them.

In the current implementation, victim list reconstruction is done by a kernel thread. This thread sleeps waiting for requests and, on being woken up with a request, does the reconstruction and sleeps again till the next request. The size of the list is fixed at 4096. A victim can be sought in either blocking and non-blocking mode. In the blocking mode, on failing to find a victim, the calling thread is suspended till the reconstruction of the victim list is done. This mode is used for invalid-RAID5 migrations which can not continue till a victim is obtained. In non-blocking mode, if no victim can be found, the victim list reconstruction thread is signalled with a request and the call immediately returns failure. This mode is used by RAID5-RAID1 migrations.

4.9 Interface to Applications

We have implemented an interface for use by privileged applications to access the data and services of the driver. The motivation for introducing this interface was to implement only mechanisms inside the kernel and offload policy decisions to user level applications. These are implemented as `ioctl()`s.

Get Maptable Address Using this `ioctl()`, an application can get the kernel virtual address of the maptable and the number of entries in it. The application can then perform an `open()` on `/dev/kmem` and `mmap()` the region into its address space. As the access frequency information is also kept in the maptable entries, the application can keep track of how often a stripe has been accessed without

making any further calls to the kernel.

Get Bitmap And Maptable Using this `ioctl()`, an application can read copies of the allocation bitmap and the maptable frozen at some point in time. The driver locks the entire maptable (by setting an interlock that spans the entire device), and then the maptable and the bitmap are copied into the user buffer and the lock on the maptable is released. Using this information, an application can compare the allocation bitmap against the maptable to find if any of the physical stripes have gone unaccounted which can happen when the system fails when a migration is in progress. The application (eg. `device-check`) can then fix this by using another `ioctl()`. Since the temperature information is also available in the maptable, the application can know the temperature without making further calls to the kernel.

Get Compressed Table Address Using this `ioctl`, the user application can access the compressed table and hence can get information about every compressed stripe, such as compressed length and physical stripe backing for this stripe.

Get Access log Using this `ioctl()`, an application can get the kernel virtual address of the access log and the number of entries in it. The application can then call `open()` on `/dev/kmem` and `mmap()` the region into its address space and keep track of the accesses without making any more calls to the kernel.

Migrate Given a RAID1 stripe and a RAID5 stripe, this `ioctl()` migrates the RAID1 stripe to RAID5 and vice versa. This `ioctl()` along with the former can be used by an application to keep track of access patterns and perform prefetching. The application can maintain its data structures in user space which do not pin memory down. To reduce the system call overhead, multiple requests can be grouped together and passed as a list. The request is failed if any of the stripes has already changed the type.

Age The maptable can be aged using this `ioctl()`. This way we implement software LRU for maintaining the access frequencies of stripes. The `accessed` bit is ORed with the `temp` field after it is shifted to right by one bit. The `accessed` bit is then cleared. Aging is also performed from within the driver during victim list reconstruction if sufficient RAID1 stripes are not found to fill the list.

Sync Meta Data This results in all meta data structures being updated to stable storage.

Punch Hole A given RAID5, cRAID5 or RAID1 stripe is marked an invalid stripe and the backing physical stripes are released.

Migrations from/to cRAID5 These `ioctls` try to migrate RAID1/RAID5 to/from cRAID5.

cRAID5 Move This `ioctl` allows us to move the cRAID5 stripe from one physical stripe to another physical stripe as backing store. This `ioctl` can be used by an user level application to take policy decisions for compaction.

Make	Quantum Fireball 1280S	
Formatted Size	1,281,982,464B	
Drive Config	Disks	2
	Heads	4
	Tracks per surface	4,142
	Sectors per track	95-177
	Bytes per sector	512
Perf Specs	Average seek(ms)	<11
	Rotational speed(RPM)	5,400
	Avg. rotational latency(ms)	5.56
	Internal Data Rate(MB/sec)	5.8-10.4
	Cache buffer size(KB)	128

Table 1: The specifications of disks used

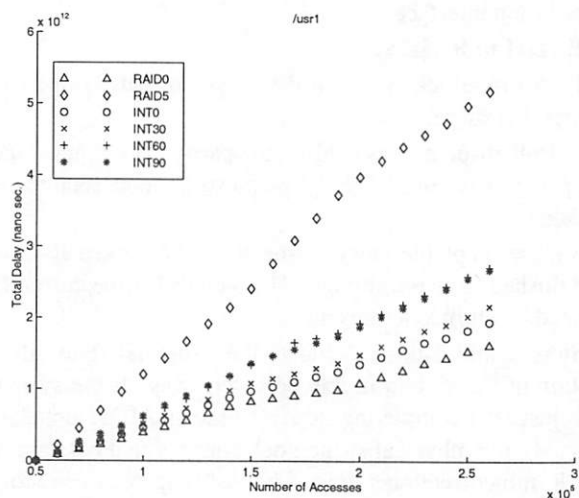


Figure 10: Total Delay vs. Number of Accesses for /usr1 disk

5 Experimental Results

We present experimental results on the Solaris platform with 2 layers (RAID1 and RAID5). We expect to provide results for Linux with 3 layers (RAID1, RAID5, cRAID5) by end of this year.

Setup The Experimental setup consists of a Sun SPARC5 workstation, with 32MB RAM. Five 1.2GB disks are connected to the machine over a 10Mbps SCSI bus. The specifications of the disks is given in Table 1.

Results and Analysis We have used HP disk traces [RW93] to evaluate the performance of our driver in comparison to RAID5 and RAID0. The trace that we used has been generated on a departmental server (snake) with accesses for two filesystems /usr1 and /usr2.

Both the integrated driver and RAID5 use 5 disks. RAID0 configuration uses only 4 disks, as it does not maintain any parity. The integrated driver is configured with 25 percent of the physical storage under RAID1. The stripe length for all configurations is 64 sectors.

Before applying the traces to our driver, we mounted a

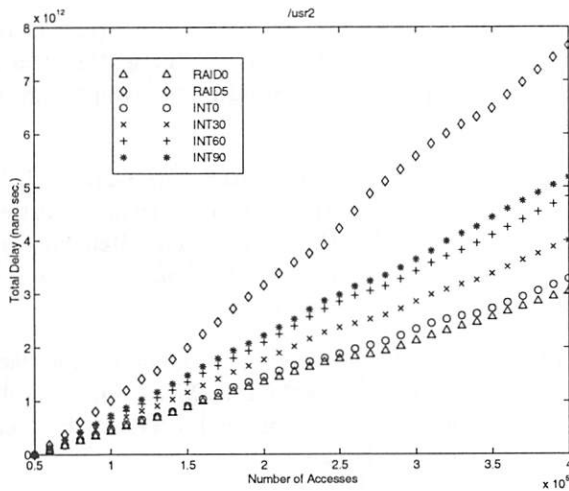


Figure 11: Total Delay vs. Number of Accesses for /usr2 disk

Disk#	Average Access times (ms)					
	RAID0	int0	int30	int60	int90	RAID5
/usr1	7.70	9.25	10.32	12.87	12.73	24.37
/usr2	8.67	9.36	11.42	13.77	14.81	21.87

Table 2: The average device access times

filesystem on it and populated it to various degrees. The first 50000 accesses were used as warmup, as the way we laid out the filesystem and populated it need not reflect the use of the traced system. The results given are for the remaining accesses. In all the figures and tables, the integrated driver is denoted by *INTx* where *x* is the degree (percentage) to which the device was populated.

Figure 10 and figure 11 show the total delay against the number of accesses for the two traces. The integrated driver performed 30-50 percent better than RAID5 even when populated to 90 percent.

Table 2 shows the average access times for the integrated device populated to various degrees along with RAID0 and RAID5. The fresh device (just ran *mkfs*) performed far better than the one that was populated. This was expected as initially all the stripes are of invalid type and writes result in the allocation of physical stripes that are contiguous which results in less seeking.

The traces exhibit high degree of locality. Figure 12 and figure 13 show the number of migrations against the number of accesses for the traces on an integrated device populated to 90 percent. The number of migrations remained very small compared to the number of accesses (<1%). Despite such high hit rates (>99%), the integrated driver access times increased with population, probably due to the following reasons:

Suboptimal data placement The present implementation has a very simple data placement policy. We simply select the next RAID1 stripe on the victim list and steal one of its

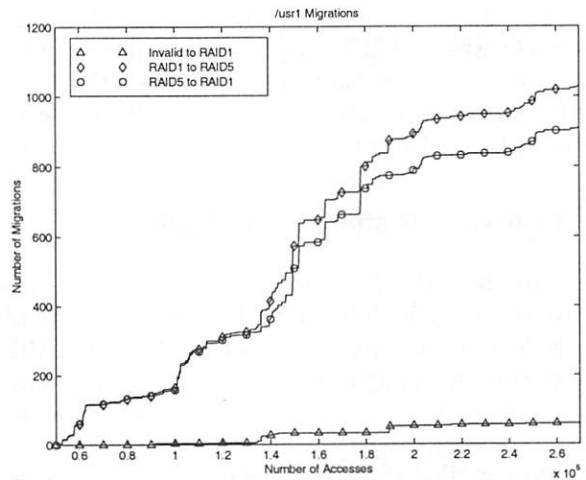


Figure 12: Number of Migrations vs. Number of Accesses for /usr1 disk

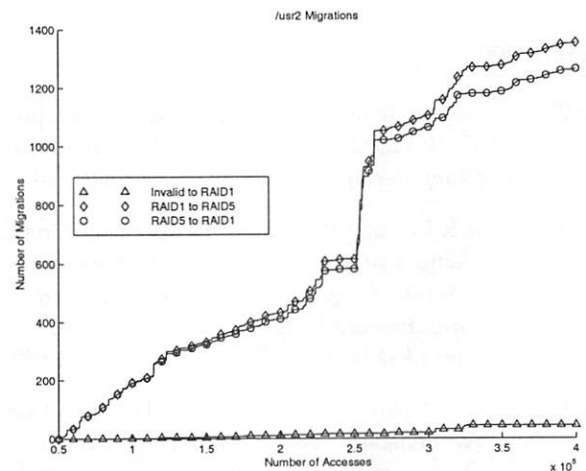


Figure 13: Number of Migrations vs. Number of Accesses for /usr2 disk

physical stripes when a RAID5 stripe or an invalid stripe needs to be migrated. A better policy should take seek distances into consideration.

High migration cost The current implementation does not maintain a pool of free physical stripes. Once the entire physical space is used up, any more migrations need to demote existing RAID1 stripes which involves updating parity, and writing metadata synchronously. This severe penalty on migrations can be reduced by making sure that there exist enough free physical stripes at all times.

6 Conclusions and Future Work

We have designed and implemented a two-tier RAID storage system using RAID1 and RAID5. We will be finishing the implementation of a 3-tier system (RAID1, RAID5, cRAID5) on Linux by end of this year. We also intend to present a evaluation of the Linux prototype at the same time.

Adding an NVRAM layer to improve write performance needs to be investigated. Prefetching and replacement of stripes across tiers is another area [CFKL95, KTPB96].

7 Acknowledgements

We thank John Carmichael, then at Veritas Software Corp., for suggesting that we look into this area, Fred van den Bosch of Veritas for his help and interest, S. Roy of Veritas for providing valuable technical assistance while implementing the driver, John Wilkes of HP Labs for providing the traces, Vaishnav Malay for his help in deciphering the traces and Kameshwari for suggestions.

References

- [AS95] Sedat Akyurek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.
- [LK91] E.K.Lee and R.H.Katz. Performance Consequences of Parity Placement in Disk Arrays *4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, Palo Alto, CA, April 1991.
- [CEJ+92] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high performance parallel storage device with strong recovery guarantees. *Technical Report HPL-CSP-92-9*, Hewlett-Packard Laboratories, November 1992.
- [CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, Kai Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling
- [KTPB96] Tracy Kimbrel, et. al. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. *USENIX 2nd Symposium on OSDI'96*
- [CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [DMAPI] Data Management Interface Group. Interface specification for the Data Management Application Programming Interface (DMAPI), version 2.1. March 1995.
- [ES92] R. M. English and A. A. Stepanov. Loge: A self-organizing disk controller. *Proceedings of the Winter 1992 USENIX Conference*, pages 237–251, January 1992.
- [HD89] Hui-I Hsiao and David J. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. *Technical Report CS TR 854*, University of Wisconsin, Madison, June 1989.
- [KL96] Thomas M. Kroegeer and Darell D. E. Long. Predicting File System Actions from Prior Events. *Proceedings of 1996 USENIX Technical Conference, San Diego, CA*, pages 319–328, January 1996.
- [MK96] Kazuhiko Mogi and Masaru Kitsuregawa. Hot mirroring: A method of hiding parity update penalty and degradation during rebuilds for RAID5. *Proceedings ACM SIGMOD 96, Montreal, Canada*, pages 183–194, 1996.
- [RW93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Proceedings of the Winter 1993 USENIX Conference, San Diego, CA*, pages 405–420, January 1993.
- [Veritas] Veritas Software Inc. Enhancing Hardware RAID with Veritas Volume Manager. Available at <http://www.veritas.com>.
- [WGSS95] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 96–108, December 1995.

Extending Internet Services Via LDAP

James E. Dutton

Southern Illinois University

jimd@siu.edu, <http://www.usenix.org/events/usenix2000/freenix/dutton.html/>

Abstract

This project report examines the use of an LDAP (Lightweight Directory Access Protocol) V2 server to provide an easily accessible data storage facility. The main purpose of the LDAP database is to store related information based on a common thread such as a person's name, an organization's name, or the description of a service offered, in a simple yet hierarchical structure.

The use of LDAP enables new fields to be added to existing user information to 1) enable end-users to store pertinent user information to be used by a mainframe-to-PC intermediary file server using Samba, 2) enable new groupings of electronic mail distributions to be created with little or no change to Sendmail, and 3) enhance the granularity of InterNetworkNews (Usenet) article submission acceptance capabilities.

Some additional benefits of these facilities included using a single, non-proprietary database which required very little new coding to make use of. The data used for the various facilities were easily associated with database objects defined for enterprise personnel. The administration load for each service was reduced since service related data, such as userids or mailboxes, were not maintained directly as a part of the specific service. The Internet Directory Service, as provided by the LDAP server, is accessible by several methods, rather than just one specialized or proprietary interface.

1 Introduction

With the advent of Internet Directory Services based on the Lightweight Directory Access Protocol (LDAP) standards, it is now easier to enhance, or extend, existing Internet Services as well as create new ones. An Internet Directory Service need not be limited only to user information for the benefit of electronic mail clients.

Three specific Internet Services were enhanced through the use of an LDAP server. They included: e-mail servers using Sendmail, Usenet or Network News

servers using InterNetworkNews (INN), and file transfer servers using Samba[14] (PC NetBIOS file system on a UNIX host) and Expect[13] (an interactive process scripting language for UNIX).

As its basic function, the LDAP server provided a data storage facility which was easily accessible using a Perl script or simple modifications to supplied application configuration code (Sendmail). This avoided developing specialized databases with specialized program interfaces. Also, no fundamental changes in each of the Services were required.

The LDAP server also provided a central data-store, much as a traditional database, where many sets of information, such as a list of authorized Network News posters for a given internal or local newsgroup, a list of Internet hostnames associated with some specific services such as electronic mail and Network News, and a list of people within certain organizational groups which may be independent of the other lists, could be kept – regardless of their association with each other. In other words, there is no need for a specialized database just for electronic mail, and another one for Network News, and yet another one for the file transfer mechanism. At the same time, the data used for these enhanced Internet Services could be, and were, related to a specific user or entity. In the particular case of Sendmail, the LDAP server replaced one of the Sendmail user databases.

Another benefit to using an Internet Directory Service was the fact that there are multiple publicly available access methods, such as Frank Richter's Web500GW Web interface[11], Kartik Subbarao's *ldaptool.html*[18] for use with Web browsers, or one of the University of Michigan's specialized, yet freely available, interface programs (i.e., Wax500, Max500, Xax500). This meant that a commercial or proprietary program is not needed to use or access the LDAP data, which means easy maintainability and reduced costs.

At the same time, access to specific data items within the server can be easily restricted without writing specialized access application programs. Using LDAP Access Control Lists (ACLs), the appropriate access permissions are established for some of the data used in

these facilities.

While each of the three services enhanced with LDAP could have provided its own facility to store and retrieve some of the user/service data required, this would have led to extra maintenance for each service. Using the Internet Directory Service method, very little maintenance is required to store and obtain the necessary data for the enhanced services. After the installation or addition of the relatively small sets of code for each service to access the LDAP data, all maintenance is then left to just the LDAP service itself.

A late addition to this project is a simple Perl script to formulate an LDAP query specifically looking for information about a LAN Administrator, and then display the result in a simple table. This required a locally installed portion of the LDAP suite (LDAPSEARCH) to perform the actual LDAP work. The Perl script's main function was to take the user's input, generate the LDAPSEARCH parameters, execute the LDAPSEARCH program, and display the selected output in a more user-friendly format.

Another late addition to this project is a simple JavaScript function for a Netscape V4 browser (which has built-in support for LDAP) that implements a simple but effective search for LAN Administrators based on one of several criteria. This is another example of a relatively easy method to extend the use of and access to data stored in an Internet Directory Server.

Lastly, this report does not discuss the implementation of network and data security, including passwords. Some general comments on security, however, can be found in Section 11.

2 Executive Overview of LDAP

The Lightweight Directory Access Protocol is an Internet standard that brings X.500 Directory services to the Internet. Implementations of LDAP provide for a structured database with entries, referred to as "objects," formed by single-word character key and multi-word/line character value pairs. A few special attributes provide for binary data values. A standard set of keys are well defined in the LDAP standard and can be considered as user friendly, for the most part. In general, the keys are usually abbreviations of an LDAP attribute name. For example, two common keys, or attributes, are *dn*, Distinguished Name (DN), and *cn*, Common Name (CN).

Many of the LDAP attributes are used to store, human readable information about people and organizations.

The attributes usually identify something specific about a person or organization such as an electronic mail address (*mail*), or a commonly used name, nickname, or pseudonym for a person, organization, or organizational unit (*cn*), or computer userid (*uid*).

The basic LDAP implementation usually provides for clear text password authentication only. This means that when an LDAP client is required to send a password to an LDAP server, that the password is not encrypted, but is sent as plain or clear text. Some specific implementations or site provided add-on programs may provide for secure client, or user, authentication using Secure Sockets Layer (SSL) or other mechanisms, but this is not yet a function of the LDAPv2 protocol itself. Authentication is the process of sending a user-identifying data string, commonly an LDAP Distinguished Name from an LDAP database object entry, and its associated password string. This is normally required only when updating an LDAP database entry. Most LDAP queries are performed without any authentication, and appear to the LDAP server as a "null" or undefined user or client. User/client authentication is independent of access control mechanisms, ACL lists (see Section 10).

Some descriptions of LDAP liken it to an electronic telephone book, or "yellow pages" directory, though that is only part of what LDAP can be used for. LDAP databases are most often organized in a tree or hierarchical structure. A large structure may be distributed over more than one LDAP server, and may include references to other LDAP servers, providing for a distributed directory service.

3 Why Choose LDAP?

Many database systems are available, so why choose an LDAP database? Without going into great lengths to examine all of the possibilities, here are some reasons for choosing LDAP versus other database or X.500 (DAP) products:

- Two well known and publicly available LDAP packages are LDAP-3.3[6] from the University of Michigan (UMich) and OpenLDAP-1.2.x[15] from the OpenLDAP Project.
- Netscape V3/V4 Directory Server is a good commercial implementation of an enhanced/updated UMich LDAP-3.3, which provides another good LDAP source that closely follows the open (IETF) LDAP standards
- some X.500/DAP servers are not user-extensible, which is not a limitation of X.500/DAP per se, but

of the particular implementations. The LDAP implementations mentioned in this paper are all user-extensible. Where a given X.500/DAP server is user-extensible, then it may be used in place of an LDAP server.

- LDAP compliance and compatibility is appearing in more and more user and networking software
- in many cases, a proprietary interface is not needed to access an LDAP database, as is true with at least some commercial database systems and some commercialized X.500/DAP databases which require a specialized access client program that works only with that particular system
- LDAP is oftentimes a lightweight database well suited for simple, lightweight, data storage that is easily accessible by many Internet-based applications; this at least partly implies that a large database system is not needed, though one could probably be used; it should be noted, however, that there have been some very large LDAP databases implemented
- data inquiries are relatively simple, especially when compared to some types of commercial database systems
- methods developed to access an LDAP database are not limited to one particular language or compiler
- as more Internet-based applications become LDAP compatible, they will increase the number of applications that can be easily integrated with other Internet-based applications using common data in an LDAP server
- LDAP provides a fairly nice data structure that lends itself well to organizing certain types of data, most of which relates to people, processes, organizations, and services
- LDAP databases are easily extendable in their data types and data structure
- small to medium sized LDAP servers can be implemented on small to medium sized computers quite easily, which may seem trivial until compared with the size of machines dedicated for large database systems
- LDAP can be implemented on commercial computer systems or free UNIX-like systems, which may lead to sizable cost benefits

4 Project Overview

The following is a brief overview of the three major and two minor uses of LDAP already mentioned. The later additions will be presented first since they are short and simple.

- a simple script to query LDAP database for LAN Admin information

Having the requisite portions of the LDAP suite installed locally, a short line-mode Perl script was developed to perform an LDAP search for the Administrator of a LAN, based on one of several types of identifiers. The purpose is to provide a simple, easy, and quick method to look up the LAN Administrator for a given LAN, providing their name, phone number, and electronic mail address.

- a simple LDAP search via a short JavaScript-based Web page

The intent here is to use an LDAP-enabled Web browser to initiate an LDAP inquiry without requiring any CGI scripts or other services from a Web server. The Web browser itself acts as the sole LDAP client. A simple HTML form is used to obtain the LDAP search item, and a simple JavaScript function then converts that into an LDAP URL[16]. When this is inserted into the browser's location field, the LDAP search is performed.

- LDAP implementation with Sendmail source and LDAP enhancements to Sendmail process

Using LDAP required two enhancements to Sendmail: using internal hooks in Sendmail for LDAP; and an LDAP-based mail delivery program, *mail500*[12, 15], external to Sendmail. With these extensions, simple virtual mail users with different address formats are easily created and serviced. Once Sendmail is installed with the required LDAP libraries, a small set of code is added to the Sendmail configuration file to enable both enhancements. In one case, the mail was handed off to the *mail500* LDAP tool, which performed the final delivery. In the other case, Sendmail itself connected to the LDAP server to find the requisite information to complete the mail delivery that it would perform.

- enhancement to INN (InterNetwork News) via a Perl script

The InterNetwork News product provides a simple security scheme to control access to newsgroups[10]. More finely detailed access methods are not directly available with the basic INN

program. Starting with version 2 of INN, a Perl "hook"[8, 9] is provided to invoke user defined code. Using this Perl script and data stored in an LDAP database, a finer grained access method is now possible. It also provides the ability to define a multi-tiered access method. In this report, this method is used to control authorized postings to a local newsgroup that did not use the network news "moderated" format and controls.

- implementing a mainframe-to-PC File Relay using Samba, Expect, and LDAP

This facility used a UNIX server running Samba, the OpenSource UNIX NetBIOS file system service, to provide a type of automated file relay between a Windows PC and a mainframe. In reality, the PC user had two network drives, an "input share" and an "output share," that appeared to connect them directly to the mainframe for simple output retrieval and job submission. LDAP stored user specific information to be used by the FTP process to the mainframe. Expect was used to automate the process of FTPing a user "job" to the mainframe for execution from the "input share," and keeping a log of the FTP process. Mainframe output was FTPed to the "output share" by some additional job control language statements that executed an outbound FTP session from the mainframe to the Samba server.

4.1 Environments Used

There were three network environments used for different stages of developing these facilities. The statistics listed elsewhere will be affected by the capabilities of the different networks and hosts listed below. Note that system loads are not reported in any of the statistics.

- development and test environment (#1)
LAN: 10Mb/s Ethernet
LDAP/Sendmail server: 50MHz Motorola 68060 CPU, 32MB RAM, 10000rpm SCSI HD, Amiga A2000 workstation
INN server: 40MHz Motorola 68040 CPU, 32MB RAM, 7200rpm SCSI HD, Amiga A2000 workstation
- test environment (#2)
LAN: 10Mb/s Ethernet
INN/Sendmail server: 400MHz Intel Pentium II CPU, 128MB RAM, 7200rpm IDE HD. Dell OptiPlex GX1 workstation

LDAP server #1: 400MHz Intel Pentium II CPU, 128MB RAM, 7200rpm IDE HD, Dell OptiPlex GX1 workstation

LDAP server #2: 70MHz Fujitsu microSPARC II CPU, 64MB RAM, 7200rpm SCSI HD, Sun SPARCstation 5 Model 70 workstation

- production environment (#3)

User LAN: 10Mb/s Ethernet

Server LAN: 100Mb/s FDDI

LDAP server: 167MHz Sun microSPARC II CPU, 256MB RAM, 7200rpm SCSI HD, Sun Ultra Enterprise 2 server

INN server: IBM RS/6000 CPU, 192MB RAM, 7200rpm SCSI HDs, IBM PowerPC Model 250 server

5 Simple Perl Script To Perform LDAP Query For LAN Admin

The LDAPSEARCH tool is a line-mode access tool that sends an LDAP query to an LDAP server, and displays all of the data returned in {key}{value} pairs. The actual LDAP search is a one-line command. The Perl script enables the user to identify something about the desired LAN Administrator such as IP subdomain name, or AppleTalk network number range. The script then calls the LDAPSEARCH tool to talk to the LDAP server and get the desired data, which was limited by a set of search command, key selectors. The results are displayed as a simple line-mode table, as shown in the example in Figure 1.

While the LDAPSEARCH tool performs the actual LDAP lookup, its command format can be very long, tedious to enter, and not pleasing to the eye to behold. Also, LDAP results are usually one or more lines of {key}{value} pairs that usually have a raw appearance. The Perl script performs all of the work to make the search process simple and effective, and provides a better display.

Figure 1 is a sample result of the *QLANADMIN* Perl script used to provide this service, searching for the LAN Administrator of a specific IP subnet. When no parameters are given, the script will display information on how to use the script, listing the various parameters accepted. Figure 2 is a sample of the the LDAPSEARCH command used and the data returned from the LDAP server.

The LDAPSEARCH command is given several "selection attributes" that it passes to the LDAP server to restrict the number of attributes that will be returned by

the server. In this script, the four selection attributes specified were: *lanadmin postofficebox telephonenumber dnsadmin*.

```
qlanadmin 216.000

=====
Subdomain      : grdsch
Network Protocol : IP
LAN Administrator: FirstName LastName
Department     : Department Name
E-mail Address  : userid@mail.host
Telephone Number : (xxx) xxx-xxxx
DNS Administrator: FirstName LastName
=====
```

Figure 1: QLANADMIN

```
ldapsearch -h ldap.hostname -b o=ournet,
c=us "subnet=216.000" lanadmin
postofficebox telephonenumber dnsadmin

dc=grdsch,dc=ournet,dc=edu,o=ournet,c=US
lanadmin=cn=LAN Admin name,
group=employee,ou=orgName,o=ournet,c=US
postofficebox=userid@mail.host
telephonenumber=(xxx) xxx-xxxx
dnsadmin=cn=DNS Admin name,group=employee,
ou=orgName,o=ournet,c=US
```

Figure 2: Raw LDAP search command and response

The distinction between IP subdomains and AppleTalk zone objects is made in the DN entry. Other attributes within each entry could have been used that are not a part of the DN. There are trade-offs to both approaches, but they won't be covered here.

As it was implemented, IP subdomains were represented by a DN which contained a Domain Component (DC) attribute for each component of the subdomain name. In the example above, the IP subdomain *grdsch.ournet.edu* has a DN that begins with *dc=grdsch,dc=ournet,dc=edu*. A representative AppleTalk zone is similarly identified, but only uses two DC attributes: one for the actual AppleTalk zone name, and one for a pseudo domain of AppleTalk.

Late note: multiple LDAPSEARCHs may, or may not, be better accommodated with *perlldap*[20].

5.1 Performance Observations

Two versions of the *QLANADMIN* script were created: one using Perl and one using REXX[19]. The test runs obtained and displayed the same LDAP data, and were run on the LDAP server in test environment #2 but used the production environment LDAP server for data lookups. "u" and "s" refer to user and system CPU usages, in seconds.

Test Performed	Perl Script	REXX Script
size	3,178 bytes	2,879 bytes
Perl Validate	0.012u,0.006s	-na-
display results	0.013u,0.014s	0.009u,0.015s

Table 1: Sample QLANADMIN Performances
Test #2 and Production Environment

6 Simple LDAP Query Via Short JavaScript Based Web Page

A simple Web page instructs the user to input part of an IP address or hostname, which is entered into an HTML text input field. Upon clicking on the Search button, the embedded JavaScript function creates an LDAP URL[16], in simplified and sample forms:

```
ldap://<LDAP server hostname>/
<LDAP search base><LDAP search string>
```

```
ldap://<LDAP server>/o=siuc,c=us??dc=<hostname>
```

The JavaScript function then changes the browser's current document URL, or "location field," to the composed LDAP URL which causes the built-in LDAP client to perform the LDAP search. Note that this requires the equivalent of Netscape V4 or higher.

The browser displays the LDAP results in a raw format as seen in Figure 3. The entire record returned by the LDAP server is displayed except for those fields that are restricted by LDAP access controls.

```
Object Class    top
                domain
                localadmin
dc              ournet
Notes          Internet network domain
                for our.network
associatedname  our.network
City           location
Organization   department name
postofficebox  mailbox@mail.host
subdomain      sub.domain.name
subnet        999.999
dnsadmin       LDAP DN (which includes
                user name)
lanadmin       LDAP DN (which includes
                user name)
creatorsname   LDAP Manager DN
modifiersname  LDAP Manager DN
createtimestamp19990519180831Z
modifytimestamp19990519180831Z
```

Figure 3: Sample Raw Web-based Data Display

Since there were only two possible choices for the LDAP search string, or target, a simple *if ...else* statement

is all that is necessary to create the dynamic portion of the LDAP URL. In the end, a fixed URL prefix was concatenated with the determined URL search target suffix obtained from the input field, for example (“+” is JavaScript string concatenation):

```
URLsuffix = "dc=" + form.SearchData.value;

document.location = sSearchURL + URLsuffix;
```

which is then inserted into the browser’s “location” field, causing the built-in LDAP client to perform the LDAP lookup.

The returned data is neither formatted nor limited, as it was in the case of the *QLANADMIN* Perl script, other than being limited by access controls on the LDAP server (see Section 10 for more information). This may allow more data to be displayed than needed or anticipated and may not be as visually appealing as desired.

Other LDAP search tools such as Web500GW, Wax/Max500, or Netscape Directory Server V3/V4 gateway (similar to Web500GW) might format the returned LDAP data differently. In some cases, they don’t – it has the same appearance as the raw format. They also may provide special functions for some of the attributes, such as providing an automated e-mail function upon clicking on the e-mail address. Any URLs in the returned LDAP data may also be turned into active URLs (i.e., clicking on them goes to the specified Web page).

This example Web page facility demonstrates a uniqueness about LDAP-enabled Web browsers in that it does not depend nor rely upon a Web server for any assistance. There is no CGI script that gets executed to receive the search data, perform the LDAP search, create a new Web page for the data returned, and send the new Web page back to the client for display. The LDAP-enabled Web browser is able to display the LDAP search result all on its own. For a very simple search tool that is easy to develop and use, the raw data format may not be pretty, but may suffice – at least for testing.

Kartik[18] has a nice example of an extensive JavaScript/HTML tool for administrative LDAP searches and database updates, also without the need for a Web server.

7 Sendmail Enhancements Using LDAP

When Sendmail Version 8[3, 4] is compiled with UMich LDAP-3.3 or OpenLDAP (or possibly Netscape Directory Server) libraries, it has direct access to an LDAP database. Sendmail[4, 5] also comes with *sendmail.cf*

configuration file examples for using this LDAP access. While Sendmail configuration programming is considered a black art in some cases, the enhancements discussed later used only a small number of additions or changes to the *sendmail.cf* file to provide the LDAP services previously mentioned.

In addition to Sendmail source code that provides access to LDAP databases, both the UMich LDAP-3.3 and OpenLDAP distributions include useful and important LDAP client contributions, one of which is related to electronic mail and Sendmail: *mail500*. *Mail500*[12] is used to provide external access to an LDAP database. Sendmail passes pending messages off to *mail500* which does all of the work of extracting e-mail addresses from the LDAP server. *Mail500* then passes the messages back to Sendmail for final delivery with the LDAP extracted e-mail addresses in place.

Sendmail can make use of many user-defined databases, which usually are either regular “flat files,” or “hash mapped” database files. Using LDAP is not necessary to provide for virtual e-mail users, but makes the process more flexible as well as eliminating constant changes to Sendmail configurations and/or databases.

To use the LDAP function from within Sendmail, an LDAP “database map” and “relay host macro” definition is added to the *sendmail.cf* configuration file, along with a few additional lines in Sendmail’s rule set #5. Using the external LDAP function via *mail500* required a one-line addition to Sendmail’s rule set #98 and a two-line “mailer” definition for the *mail500* external “mailer” program. The *sendmail.cf* “rule sets” are the means by which Sendmail determines what to do with any given piece of mail.

Each set of additional lines defined a specific character string which when matched to a supplied e-mail address, would invoke the respective LDAP-enabled service. Many trigger combinations are possible, but this implementation limited itself to just the two sets detailed later. The intent is to provide a simple yet effective means of selecting e-mail addresses from a fixed set of virtual e-mail users.

Virtual users, in the context of this facility, are considered to be users whose e-mail addresses map to the local Sendmail server but whose userids (left-hand portion of each of the e-mail addresses) are not found in the Sendmail server’s password file. With the modifications mentioned in place, if Sendmail fails to resolve a local userid via its password file, it then calls upon the defined LDAP server to retrieve an e-mail address from the LDAP database. In this manner, many users who have an LDAP entry based on a userid or other attribute which matches the left-hand portion of an e-mail address, and

who have an LDAP e-mail attribute defined, can be considered as local to the Sendmail server.

With the aid of LDAP, the number of the e-mail users, their identities, and their final e-mail addresses to be used for mail delivery need not be known in advance by Sendmail, thus obviating work required to create "static" user definitions for special purposes. No additional Sendmail configuration is necessary other than what has already been mentioned. Nor is any additional Sendmail configuration necessary when virtual e-mail users change, so long as the required LDAP attribute names do not change.

In Sendmail's rule set #5, one rule compares the given e-mail address with a specific format that separates the left-hand side of the address from the rest. This rule set is used to perform some specific tests on e-mail addresses that are determined to be local to the Sendmail server. One part of it attempts to locate the userid extracted from the e-mail address in the mail server's password database. This occurs in the section labelled as, "send unrecognized local users to a relay host." If the userid is not found, it will later be passed to another Sendmail rule which will attempt to send the pending mail to another mail server, defined as a mail relay.

This implementation inserts a Sendmail rule after the above section to then attempt to look up the userid in the defined LDAP database. Here, the modified Sendmail program itself is the LDAP client. If the database search returns an e-mail address, Sendmail then uses it in place of the original address and eventually attempts to deliver the pending mail to the new e-mail address. This provides for a virtual user who is not directly associated with or defined on the host running the Sendmail server but whose e-mail address conforms to the normal format for that particular server.

In rule set #98, a new rule was added to select processing by the *mail500* program. This rule set is part of the Sendmail logic used to determine how the actual mail delivery will be completed, and by which program. As a part of this processing, an external program, such as *mail500*, may be selected to complete the mail delivery.

In this implementation, the rule added to rule set #98 selects mail based on the following pattern, which matches one of the LDAP *cn* entries as shown in Figure 4:

```
<firstname>.<lastname>@<Sendmail server E-  
mail domain>
```

Now, mail with the indicated address format will be directed to *mail500* for a simple LDAP CN attribute lookup based on the *<firstname>.<lastname>* portion. If a corresponding *mail* attribute is found, *mail500* will then formulate a new address and pass the

mail back to Sendmail for final delivery. This is expected to return a single e-mail address for one person.

Another rule, added later on to rule set #98, selects mail based on the next pattern to extract multiple e-mail addresses associated with a mailing list.

```
<mailing list name>.mlist@<Sendmail server E-  
mail domain>
```

The rule strips off just the mailing list name, and passes it to *mail500*, along with the contents of the message. *Mail500* then searches for a CN attribute belonging to an LDAP *rfc822MailGroup* object that matches the mailing list name. If found, it then scans the LDAP object for all *mail* and *member* attributes, using the corresponding e-mail addresses to form a single mailing list. When all of the addresses are found, *mail500* passes the message back to Sendmail with the list of new addresses, and Sendmail finishes the mail delivery, assuming no other matching process occurs with the new e-mail address list.

Part of an LDAP entry that works with these methods might look something like the following:

```
dn: cn=Jim Dutton, ou=<deparment>,  
    ou=People,o=<our domain>, c=US  
objectclass: ...  
cn: Jim Dutton  
cn: jim.dutton  
cn: Jim_Dutton  
cn: jimd  
sn: Dutton  
uid: <userid>  
mail: <userid>@<hostname>
```

Figure 4: Sample LDAP Entry For Sendmail Use

For the LDAP database lookups this facility chose to use the LDAP Common Name (CN) attribute to search for an entry that would have a *mail* attribute defined. The left-hand side of the e-mail address is extracted via the applicable Sendmail rules. This is then used as the key value for the LDAP CN search. The LDAP search will then return the value of the *mail* attribute if the corresponding CN attribute is located and it has a *mail* attribute defined. Another LDAP attribute could have been used to provide the LDAP search key, but it must be related to an LDAP entry which will return a valid e-mail address.

With LDAP, some attributes can have multiple instances whereas others normally cannot. The Common Name attribute is one that is normally allowed to have multiple instances, or occurrences, of attribute values. Since this does not require any special LDAP configuration to use, it is easy to make use of in this facility, and in other situations. In the above sample, multiple CN

attributes are defined including one with the specific `(firstname).(lastname)` format. This provides for multiple versions of a "name" attribute which can be used to locate a specific LDAP object, which doesn't necessarily have to be a person.

While an LDAP server also provides for internal substring matching of search keys, thus providing for entry matches based upon part of an attribute value but which can be quite costly to perform, using multiple CNs provides for a set of easily matched and defined qualifiers that may be considered alternative spellings of a primary Common Name. The use of multiple CNs, and some other attributes, increases the probability of an LDAP search hit including the possibility of providing for misspellings of the primary attribute. This becomes very apparent when a user's name as used for the LDAP DN is not the same as the user's name as used in practice. Without additional search qualifiers (i.e., multiple attributes), searches for the specified DN may indeed become difficult and frustrating. This is one of the important benefits of the LDAP data structure and usability.

7.1 Performance Observations

Test Performed	LDAP Lookup	Mail Delivery
1-user LDAP	≤ 2 secs	2 secs
1-user direct	-na-	≤ 1 sec
5-user list	< 3 secs	5 secs

Table 2: Sample Sendmail Performances
Test #1 Environment

The 1-user note size 902 bytes, including all SMTP header records and one text line. The second test was sent directly to the same mailbox acquired in the first test from LDAP. The 5-user mailing list was directed to 2 users with *mail* attributes and 3 users with *member* attributes in the mailing list LDAP object. The *member* attributes were subsequently looked up in the LDAP database for their related *mail* attributes. The mailing list note delivered was 868 bytes in size, including all SMTP header records and one text line. The mail log entry for the mailing list contained extensive debug statements that would not normally be present.

8 INN Enhancement Using LDAP

InterNetwork News (INN) is server software used to provide Network News, or Usenet, services. The host which provides this service is usually referred to as the news server. In many cases, an Internet Service Provider (ISP)

will automatically provide Network News service to new customers, giving them Read and Post access to the all of the newsgroups, sometimes also known as "discussion groups," that the ISP provides via their news server.

For the most part, the general newsgroup article reading and posting access privileges are sufficient, and no additional access mechanism is necessary. Combinations of the fields in the INN access security file, *nnrp.access* (see Figure 5), provide the basic capability to allow or disallow access to a newsgroup or the news server itself, and allow or disallow reading and posting of a news article. In some cases, however, this simple security scheme is not sufficient to enable more complex access criteria, or to provide for other qualifiers for access control.

Version 2 of INN introduced a Perl script called by *nnrpd*, the server program that handles users newsgroup accesses. It is referred to as a "hook" since the *nnrpd* program automatically executes the specific Perl program named *filter_nnrpd.pl*, if it exists. The function of this hook is to allow the local news administrator the opportunity to code whatever conditions are necessary for establishing a user's right to post an article, thus providing an extended access control function. With the basic INN user access control, article posting is either "yes" or "no" from the time the user connects to and is accepted by the news server. With the *filter_nnrpd.pl* Perl hook, the news server can now say "maybe" to an attempt to post an article, and then make a more informed decision based upon the final outcome.

INN uses a file named *nnrp.access*, whose syntax is described in [10] and illustrated in Figure 5, to control access to newsgroups. Newsgroups may be defined as "moderated" or "not moderated." Most generally available newsgroups are of the "not moderated" variety. To be a "moderated" newsgroup means that article submissions are sent to one or more persons via electronic mail, rather than being posted to the specified newsgroup. These "moderators" then review the article-to-be-posted, and if they accept it, complete the posting process (which will not be discussed here).

The goal was to provide a local newsgroup, one that is not distributed across the world, which had three special needs. The first was to allow any user with access to the newsgroup the right to post a new article (i.e., one that has no previous references). This fits into the basic INN access method whereby "Read" would be included in the *permissions* field of *nnrp.access*. However, the basic INN "Post" permission does not provide for any distinctions about what, who, or when an article may be posted. The permissions of "Read Post" allow anyone who can access the newsgroup to read any existing article, and post any desired article at will to that news-

group, without any limitations. For a new article, the basic INN access permissions are just fine.

```
<F#1>:<F#2>:<F#3>:<F#4>:<F#5>

<F#1> = IP hostname or address of user's
        IP host; may be empty

<F#2> = newsgroup access permissions
        R(ead) P(ost); may be empty

<F#3> = username user will be known as when
        using this service; may be empty

<F#4> = password user will use when using
        this service; may be empty

<F#5> = list of newsgroups accessible for
        this particular user/host
```

Figure 5: INN *nnrp.access* file

The second special need was to implement selective posting of an article that referred to a previous article, usually referred to as a “followup.” The *filter_nnrpd.pl* hook is used to make the final decision of whether to accept the article or not. Specific project instructions were given that identified only a select handful of people who were given the referenced article (followup) posting permission. They and they alone would be able to reply to an existing article with a followup article, and were to be defined as “authorized reply posters.” All other users would be denied this privilege.

The third special need was to allow authorized reply posters the ability to post followup articles without having to use the “moderated” control scheme, which is the basic INN method to control article posting. This need was taken care of as a side affect of the solution for special need #2.

An LDAP database is used to store the list of authorized reply posters, including information used for security purposes, and other information about the special newsgroup. When the news server receives an article from a user to be posted, it invokes the *filter_nnrpd.pl* hook. If the pending article is destined for the special local newsgroup, the Perl script uses the LDAPSEARCH tool to obtain the authorized reply poster list and their associated security data. If the pending article is a new article, it is automatically accepted. If it is a reply, or followup, article to the special local newsgroup, the article author is verified against the authorized reply posters list and their associated security data. If there is a match, the article is accepted and subsequently posted.

The choice of using an LDAP database is mostly to give the owner of the special local newsgroup the ability to make changes to the authorized reply posters list without requiring any changes to INN (including the Perl

hook). It also is meant to take advantage of existing LDAP entries for the individuals responsible for the local newsgroup, and entries for locally provided services (i.e., Network News). Using LDAP also means that the news administrator does not need to perform any ongoing maintenance to INN or *filter_nnrpd.pl* to keep the list up to date. It also provided ancillary information about the authorized reply posters that would not have been possible with just a hard-coded userid in *nnrp.access*.

The *filter_nnrpd.pl* script, as implemented for this facility, is set up with a Perl array to specify more than one LDAP server. This assures access to an LDAP server in the event that one or more of the servers defined is unavailable for service. Required LDAPSEARCH command parameters and LDAP search parameters are stored in Perl variables to make LDAPSEARCH command execution simpler to code. A local log writing facility, *logger*, is called upon to enter lines in the *news.notice* log file to show important operations of the script.

To obtain the list of authorized reply posters, the LDAP search base is set to the LDAP Common Name of the object representing the special newsgroup. The list of attributes to return is assigned to the *attr* variable as,

```
$attr = "certifiedAuthor";
```

and the LDAP search is performed, using the following Perl statement,

```
@result = `ldapcmd $ldaphost[$i] $parm
           $scope "$base" "$filter" $attr`;
```

where predefined Perl variables are used for the command and its parameters. The LDAP search results are put into the *result* array, which is then searched for the desired information. Access to the data is controlled by the LDAP ACL as shown in the first ACL example in Section 10 (Figure 8) where the LDAP server receives a network connection from the news server and matches its IP address against the *by addr* portion of the ACL. Using information obtained from the pending news article and the LDAP server, the script determines whether the article will be accepted for posting.

Using the Web500GW[11] Web interface, the special newsgroup owner can readily see and change the LDAP data relevant to their newsgroup. They are then able to manage the authorization list without any further work by the news administrator. With the Perl hook and coding added to access the LDAP database, simple or complex decisions can be made with regards to specific article postings. This then extends the levels of security normally available to INN without requiring local modifications of the INN programs themselves.

8.1 Performance Observations

While not exhaustive, the following simple data may help to show some of the performance differences between using an INN server without any LDAP enhancements, and an INN server using the enhanced *filter_nnrpd.pl* filter with multiple LDAP searches. The “Prod. Script” uses LDAP and extra coding whereas the “Orig. Script” that comes with INN basically does nothing.

Test Performed	Prod. Script	Orig. Script
size	14,091 bytes	1,207 bytes
Perl Validate	0.548u,0.233s	0.127u,0.180s
One line Post-s	0.975u,0.747s	0.404u,0.363s
One line Post-u	≤ 6 secs	< 0.75 sec
NNRPD write/log	< 4 secs	-na-
NNRPD reject/log	≤ 6 secs	-na-
LDAP search/log	≤ 3 secs	-na-

Table 3: Sample INN/LDAP Performance
Test Environment (#1)

Test Performed	Prod. Script	Orig. Script
size	14,100 bytes	1,207 bytes
Perl Validate	0.180u,0.060s	0.010u,0.050s
One line Post-s	0.400u,0.380s	0.180u,0.280s
One line Post-u	≤ 2.25 secs	< 0.5 sec
NNRPD write/log	≤ 2 secs	-na-
NNRPD reject/log	≤ 2 secs	-na-
LDAP search/log	< 2 secs	-na-

Table 4: Sample INN/LDAP Performance
Production Environment

The “u” and “s” symbols in the times represent user and system CPU usages in seconds, respectively. Those entries which are not applicable since the script does not perform or have any impact on the indicated function are identified with “-na-.”

The Perl validation is the simple *perl -c* test which validates the Perl code (i.e., performs the Perl “compile” operation) which is affected by the size of the code being tested and the complexity of the code. The original script contains only a few active lines – the remaining being either blank or commented out.

The “One line Post” times come from a new article posted to the special test newsgroup that contained one line of text, which varied slightly in size, plus the normal article header records. The “-s” times are news server times for just the *nnrpd* program itself. The “-u” times represent the real-time measures of time from when the

user initiates the posting action (i.e., presses the Post button, or equivalent), and the article is posted (as reported by the user’s news client).

All “/log” entries represent the elapsed time observed for the particular function performed, reported as integer values. Logging is accomplished with the system *logger* command, where each log line is a separate call to *logger*. Time values indicated as “less than” are strictly less than the value indicated. Values indicated as “less than or equal to” are predominantly less than the value indicated – there were a small number of observations that did equal the specified value.

“NNRPD write/log” times represent the total elapsed time for *filter_nnrpd.pl* to write its entries to the log file after the user posts the article (i.e., presses the Post button, or equivalent), and *filter_nnrpd.pl* accepts it.

“LDAP search/log” times are taken from an LDAP server log which show the total elapsed time for the LDAP server to accept a connection from the news server and process its LDAP search requests for the special newsgroup entry, return the data to the news server, and close the network connection.

“NNRPD reject/log” times are from the news server log which show the total elapsed time for the *filter_nnrpd.pl* “hook” to make its posting failure determination and write out its log lines.

The differences in log file writing times are affected by the throughput of the associated network and the speed of the associated servers, which is to be expected. The different environments, slower test, faster production, also represent a difference in the relative “size” of the servers (i.e., small versus medium). The amount of *filter_nnrpd.pl* logging appears to be the largest delay factor in this process, all other things being equal. It should be noted that *filter_nnrpd.pl* can operate without generating any log entries, but for administrative purposes, ten or so lines were normally logged.

Overall, while increasing news server system time for log writing, Perl “hook” processing, and external service (LDAP) processing, the amount of additional time for the LDAP-enhanced INN services is reasonable. Turn off all *filter_nnrpd.pl* logging and the additional time can be further reduced, making it negligible.

9 Implementing a PC-to-other host File Relay

The main goal of this facility is to provide an easy way for a PC user to submit jobs to a remote host and obtain output from a remote host, without the PC user having

to connect or logon to that remote host. All of the file transfer exchanges appear to be local to the PC user. This eliminated the need for the PC user to stop what they are doing, initiate an FTP process to the remote host, and then issue one or more commands to send or receive a file. Occasionally, a login or telnet session would also be required to obtain the desired output. All they need to do now is to copy a file from or to two network drives (remote Samba server "shares"), and everything else is done automatically.

While this particular facility was developed for use with an IBM mainframe and NetBIOS/Windows PCs, it can be equally used for other hosts where a remote job could be executed and file transference is accomplished by FTP (or some other suitable function). Implementing NetaTalk[21], the UNIX-based AppleTalk file system, this file relay is also available to Macintosh hosts. Again, while the remote host details will be specific to a mainframe in this project report, the process is not limited to such a host. The same is true for the user side. In the remainder of this section, the terms "PC user" and "PC host" are interchangeable with other user-based scenarios using different types of computers.

The key mechanisms used for this file relay are FTP and user workstation file copy. The remote host is expected to be an IP host which provides an FTP server to allow files to be sent to it. In the case of the IBM mainframe, a special FTP command (SITE) was used to direct the incoming file to the "job queue" of the remote host. To obtain remote host output, the user would add several FTP commands to their "job" which executed on the remote host. This would send the output data to the Samba server. To obtain a local copy of the output data, the user simply copies the file from their Samba server "output share" to a local folder.

It should be noted that the LDAP service was chosen to store the user's remote host FTP/login password, but the same data can be supplied in the user's job file. In general, however, it is simpler to ask an LDAP server for a data entry than to parse a job file, looking for the same data even if a fixed scheme is applied and expected in locating said data. The LDAP mechanism also allows more data to be stored for the given user, such as a "notify" field intended to be used to send a notification to the user in certain situations. Again, this same information could be incorporated into the job file to be submitted, which then must be parsed and properly extracted. The LDAP mechanism helps to reduce the multiple possibilities of user data entry errors, though it certainly does not eliminate all possible user errors.

Figure 6 and Figure 7 illustrate the before and after affects of providing this service.

In Figure 6, the user must manually go to the remote host to submit a job or retrieve job output. This may or may not require a telnet session to find the job output information. The user must be familiar with the FTP process including all special commands such as the SITE command required in our situation. They must also be familiar with finding their output data files and how to move them to where they want them, most likely using an FTP session. For some users this process may be counterintuitive and time consuming.

In Figure 7 the user has established two network drives available to them from the Samba server, which appear as folders on their local system. They can then copy files in to and out of these network drives as easily as they would with any other local folders. This provides a well understood mechanism for the user once they understand which files go into or can be found in which remote folder. They also are free to use whatever local tools they desire to do the file transference required.

The job to be run is copied into the remote submit folder on the Samba server. An automatic process added to the Samba server, using Expect, then sends the job file to the remote host for execution via FTP. Two log files are maintained: one in the user's submit folder which shows the submission process result, and one in the server's log directory which shows a count of all jobs submitted per user.

A special "job step" can be used to transfer output data to the user's remote output folder on the Samba server. This can be added to a job file placed in the submit folder, or can be a job that the user has permanently stored on the remote job execution host. Upon execution of this special job step, the remote host deposits the specified output file into the user's output folder. The user can then copy the output file to their PC.

Using this process, the user never has to logon or connect to the remote host.

10 Brief Comments On LDAP ACLs (Access Control Lists)

Netscape Directory Server, UMich LDAP-3.3, and OpenLDAP all use a scheme to control access to parts of the LDAP database by specifying one or more internal database attributes, or external real-world attributes. Netscape has renamed ACL to ACI, but the mechanism is basically the same. The biggest difference between Netscape's ACI and LDAP ACL is where the actual control list resides. With Netscape's Directory Server, the ACIs are entries within the LDAP database, usually at the root of the database tree to be controlled.

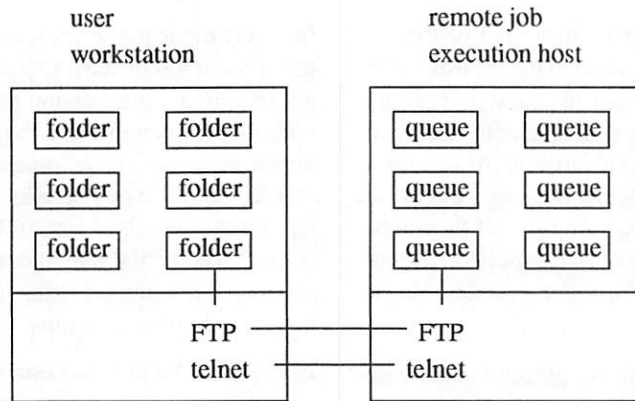


Figure 6: User to remote host via FTP and telnet
User does all of the work

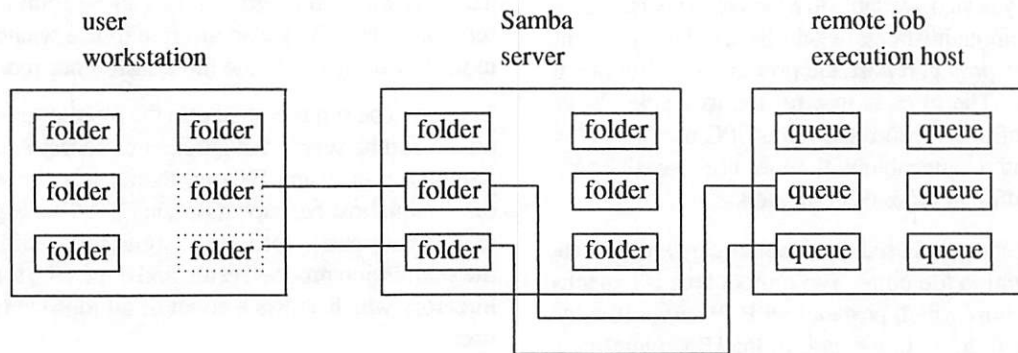


Figure 7: User to Samba server
Samba server does all of the work

With UMich/OpenLDAP, the ACLs are usually stored in an external file which is included in the LDAP server startup file.

The basic LDAP ACL definition[7, Section 5.3] is shown below. Netscape's ACI has additional attributes.

```
<access directive> ::= access to <what>
                        [by <who> <access>] +

<what> ::= * | [dn=<regex>]
              [filter=<ldapfilter>]
              [attrs=<attrlist>]

<who>  ::= * | self | dn=<regex> |
              addr=<regex> |
              domain=<regex> |
              dnattr=<dn attribute>

<access> ::= [self]none | [self]compare |
              [self]search | [self]read |
              [self]write
```

Samples of ACLs used in the facilities of this report, with some specifics replaced by generic qualifiers, are

shown in Figure 8.

```
access to filter="cn=<newsgroup name>"
      attr=certifiedHost,certifiedAuthor
by self write
by dnattr=owner write
by dn="cn=Manager,ou=People,
  o=<our domain>,c=US" write
by dn="cn=<News Server IP hostname>,
  ou=Network Hosts,o=<our domain>,
  c=US" read
by addr=<News Server IP address> read
by * none

access to attr=MVSpasswd
by self write
by dn="cn=Manager,ou=People,
  o=<our domain>,c=US" write
by addr=<Samba server IP address> read
by * none
```

Figure 8: Sample Project ACLs

The first ACL restricts all access to the *certifiedHost* and *certifiedAuthor* LDAP attributes defined in the LDAP entry for the special newsgroup whose DN begins with

the Common Name value of <newsgroup name>. Write, and read, access are granted to the person identified by the *owner* attribute, which is a “pointer” to another LDAP entry (DN). The person associated with the (directory server) Manager DN is given the same access permission.

Access for these privileges requires LDAP password authentication, where the password is stored in the LDAP entry of the authenticating user. The news server using the LDAP database is allowed to read the restricted attributes by either authenticating as the news server LDAP entry DN, or by its IP address matching the value in the *by addr* specification. All other access attempts for the two specified attributes is denied.

The second ACL in Figure 8 restricts access to the *MVSpasswd* attribute defined in any LDAP entry. The previous ACL was limited to a single LDAP entry. Here the user (*self*) identified by their entry’s DN and the LDAP administrator both have write and read access. The Samba server which uses the *MVSpasswd* data is allowed only read access, and its IP address must match that as given in the *by addr* specification. All other access attempts are denied.

After the owner of any restricted LDAP entry becomes familiar and comfortable with the process, the LDAP administrator may or may not be removed from the specific ACL. It is usually included initially to get the new service operating, and to provide a fall back in case the owner has problems updating their restricted data.

11 Conclusions and Comments

While many uses of LDAP data and servers today relate solely to electronic mail addresses, one of the inherent benefits of using LDAP is the ability to easily relate many other informational items with any given LDAP entry. This can greatly expand the knowledge base associated with a person, group, service, network entity, organization, or other type of object. This is not to say, however, that LDAP is a ready replacement for all uses or forms of other databases.

One major difference between Sendmail’s LDAP client capability and the *mail500* LDAP client program is that Sendmail only accepts one returned e-mail address whereas *mail500* accepts all returned e-mail addresses. This is not a weakness in Sendmail’s LDAP client, but rather a difference in philosophy which can be put to good use. If a note to a mailing list could generate many e-mail addresses, then allowing an external program such as *mail500* to do all of the work in assembling those addresses from an LDAP database relieves

Sendmail of work that it doesn’t really need to do, freeing it to do the work it is more expected to do – deliver mail. On the other hand, a single user LDAP lookup fits in quite well with Sendmail since it can, and sometimes does, the same kind of process in other user databases created for Sendmail specifically. It should be noted that the specific uses of LDAP in this report are not the only way that the desired affects could have been achieved.

In general, data stored in an LDAP server that is meant to be user consumable is in clear text – not encrypted. Internal server information may or may not be encrypted, but the user does not see and usually cannot access such data. Also, most general LDAP connections today are insecure, not only for connection or update authentication, but also for data transmission. Additionally, there may be possibilities of IP hostname or address spoofing. These can be serious concerns for some sites or individuals.

In the case of the file relay facility, some possible methods to increase security are:

- encrypting user’s remote host login password in the LDAP database,
- encrypting data transferred between the LDAP server and the submission client,
- using a Kerberos-enabled FTP client in conjunction with a Kerberos-enabled FTP server.

However, in the end, the file relay service is dependent on trust and accountability no matter what security mechanisms are used since its job submission client must deal with a remote host userid and password, or equivalent, to perform its function.

The facilities in this report were not implemented with the intent of providing secure applications. They are meant to show the feasibility of certain LDAP-enhanced applications, and to provide a test bed for exploring such LDAP add-ons.

Version 3 of the LDAP protocol specifies a secure BIND capability through the use of Simple Authentication and Security Layer (SASL), RFC 2222. Several INTERNET-DRAFT documents propose the use of Transport Layer Security, RFC 2246, as a means of protecting data transmitted between an LDAP client and server. It may be possible to create “secure tunnels” or “wrappers” using OpenSSH, OpenSSL, SSLeay, and/or *stunnel*[22] for LDAP clients or servers without security capabilities.

Perhaps in the future, LDAP clients and servers will have the requisite hooks built-in or coding added to provide

security at all levels of interaction. In the meantime, network, password, and data security is left up to the implementor and/or administrator. Different layers of security and some security mechanisms are discussed in [17, Chapter 11]. Actual implementation, however, is still up to the site.

Availability

Additional comments about minor problems and observations about implementing each of these facilities along with coding samples will be made available by request via e-mail. Also see the Usenix Web page in the author section.

Acknowledgements

I would like to thank Rob Kolstad for his valuable editorial assistance with this report. I would also like to thank Chris Demetriou for his valuable assistance in getting this report into its final form.

References

- [1] Timothy A. Howes and Mark C. Smith, *Programming Directory-Enabled Applications with Lightweight Directory Access Protocol*. Macmillan Technical Publishing, Indianapolis, Indiana, 1997.
- [2] Timothy A. Howes, *The Lightweight Directory Access Protocol: X.500 Lite*. Center for Information Technology Integration, University of Michigan, Ann Arbor, Michigan, 1995.
- [3] Bryan Costales with Eric Allman, *Sendmail*. O'Reilly & Associates, Inc., <http://www.oreilly.com/>, second edition 1997.
- [4] *Sendmail: src/README*. Sendmail Consortium, <http://www.sendmail.org/>, V8.[89].x.
- [5] Booker Bense, *Using LDAP with sendmail.8.[89].x*. <http://www.stanford.edu/~bbense/Inst.html/>, January 2000.
- [6] *UMich LDAP-3.3*. University of Michigan, <http://www.umich.edu/~dircsvcs/ldap/>, 1996.
- [7] *The SLAPD and SLURPD Administrator's Guide, Release 3.3*. University of Michigan, <http://www.umich.edu/~dircsvcs/ldap/>, 1996.
- [8] *INN-2.1: README.perl.hook*. Internet Software Consortium, <http://www.isc.org/>, 1998.
- [9] *INN-2.1: samples/filter_nnrpd.pl*. Internet Software Consortium, <http://www.isc.org/>, 1998.
- [10] *INN-2.1: man(nnrp.access)*. Internet Software Consortium, <http://www.isc.org/>, 1998.
- [11] Frank Richter, *Web500GW*. <http://www.tu-chemnitz.de/~fri/web500gw/>, V2.1b3, 1998.
- [12] *LDAP-3.3: mail500/README*. University of Michigan, <http://www.umich.edu/~dircsvcs/ldap/>, 1993.
- [13] Don Libes, *Exploring Expect*. O'Reilly & Associates, Inc., 1st edition, 1994. [also, <http://expect.nist.gov/>]
- [14] John D. Blair, *Samba - Integrating UNIX and Windows*. Specialized Systems Consultants, Inc., Seattle, Washington, 1998.
- [15] *OpenLDAP-1.2.7*. OpenLDAP Foundation, <http://www.openldap.org/>, September 1999.
- [16] Tim Howes and Mark Smith, *An LDAP URL Format*. RFC 1959, June 1996.
- [17] Timothy A. Howes, Mark C. Smith and Gordon S. Good, *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing, Indianapolis, Indiana, 1999.
- [18] Kartik Subbarao. *ldaptool.html* <http://developer.netscape.com/viewsource/subbarao.ldap/subbarao.ldap.html/>, September 1999.
- [19] Ian Collier. *REXX-IMC-1.7*. <http://users.comlab.ox.ac.uk/ian.collier/Rexx/rexximc.html/>, February, 1999.
- [20] Clayton Donley, Netscape Corporation, *perldap*. <http://www.mozilla.org/directory/>, 1999.
- [21] Adrian Sun, *netatalk-1.4b2+asun-2.1.3*, <ftp://ftp.cobalt.net.com/pub/users/asun/release/>, [also: <http://www.umich.edu/~rsug/netatalk/>], February 1999.
- [22] Michal Trojnara <mtrojnara@ddc.daewoo.com.pl>, Adam Hernik <adas@infocentrum.com>, Pawel Krawczyk <kravietz@ceti.com.pl>, *stunnel-3.4a*. <http://opensores.thebunker.net/pub/mirrors/stunnel/>, December 1998.

MOSIX: How Linux Clusters Solve Real World Problems

Steve McClure
smcclure@emc.com

Richard Wheeler
ric@emc.com

*EMC² Corporation
171 South Street
Hopkinton, MA 01748*

Abstract

As the complexity of software increases, the size of the software tends to increase as well, which incurs longer compilation and build cycles. In this paper, the authors present one example of how clusters of Linux systems, using the MOSIX extensions for load monitoring and remote execution, were used to eliminate a performance bottleneck and to reduce the cost of building software. We present a discussion of our original software development cluster, an analysis of the performance issues in that cluster, and the development and modifications done to MOSIX and Linux in order to produce a solution to our problem. We finish by presenting future developments that will enhance our cluster.

Introduction

As computers increase their processing power, software complexity grows at an even larger rate in order to consume all of these new CPU cycles. Not only does running the new software require more CPU cycles, but the time required to compile and link the software also increases.

EMC Corporation [EMC] is a leading vendor of storage solutions. EMC's products are more than just cabinets full of disks and tapes: they rely on a large amount of software, both inside and outside the cabinets. Each member of our development group needs to compile and link millions of lines of code on a routine basis. Building all versions of this code on an individual developer's desktop system required approximately two hours (7,200 seconds). Spending

this much time waiting for code to build is unacceptable.

One way to reduce this build time is to give each developer a higher-powered desktop computer, with better processors, huge amounts of memory, and locally attached storage. However, with dozens of developers, this is not a cost-effective solution. A variation on this is to use a very large, centralized build server, although this class of machine is extremely expensive as well.

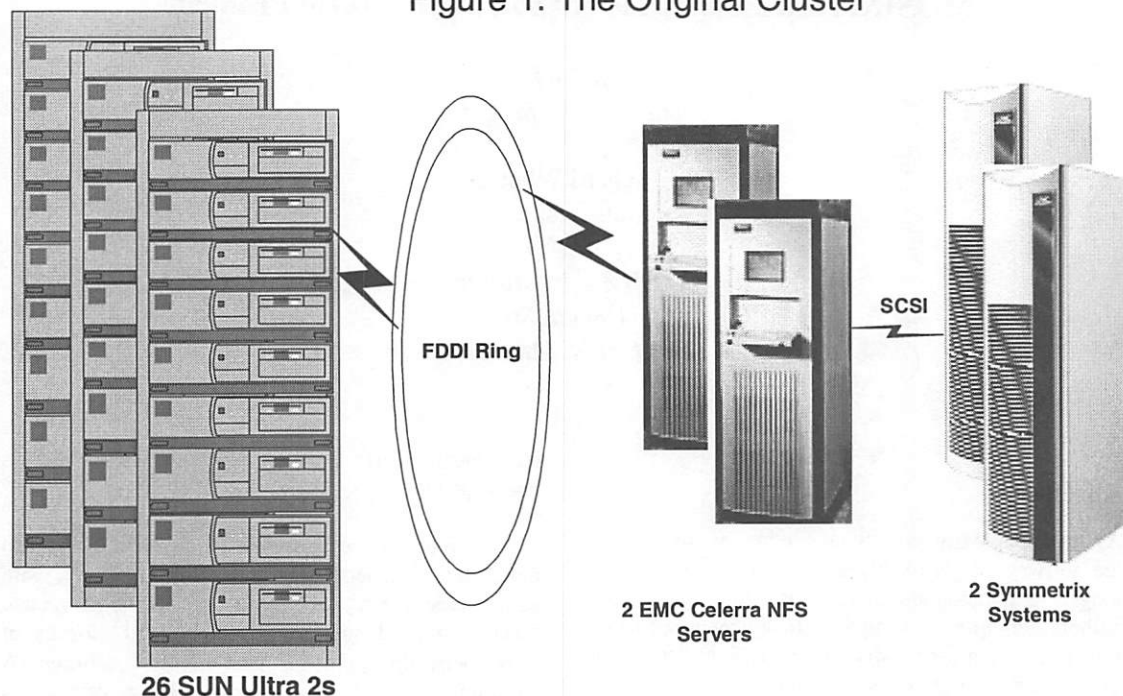
The last possibility is to use a number of smaller computers as a build cluster, where each is attached to some common storage pool so they can all operate on the same source code base. The challenge in this solution is to distribute the compilation jobs across the cluster members evenly, insuring consistent results regardless of the node on which a process runs.

The Original Cluster

EMC uses a few small clusters of computers connected to NFS file servers as our software build environment. As the size of our group and our builds grew, our existing clusters gradually became overloaded when more than two developers initiated builds concurrently. Upgrading the nodes helped somewhat, but we needed a more cost-effective solution.

This original cluster (see Figure 1) consisted of 26 SUN Ultra 2 workstations, each with 512MB of RAM and dual, 300MHz UltraSparc-II processors each with 2MB of L2 cache. The backing store was a pair of EMC Celerra NFS file servers, each connected to an EMC Symmetrix for disk storage. Each Celerra contains 14 NFS data movers, each with an

Figure 1: The Original Cluster



independent connection to the network. The network was a FDDI ring connecting all workstations and NFS data movers. Each user was given a single Symmetrix volume mapped to a single physical disk, accessible through one of the NFS data movers.

Our original cluster software was Platform Computing's Load Sharing Facility (LSF) version 3.1 [Platform] running on top of Solaris. The LSF solution is based on GNU make(1), which spawns parallel makes across the cluster, as well as some proprietary user-level software.

On the original, idle cluster, a typical user could build a complete set of binaries in ~13.5 minutes (813 seconds). This is a good improvement over the original two hours, but the real problem appears when many users want to build at once.

When two users run jobs on the cluster at the same time, performance degrades severely as the cluster starts to run out of resources and every node becomes too busy to accept new jobs. In this case, each user's build takes 22 minutes (1324 seconds) to complete. In addition, if subsequent users try to start new jobs, they are locked out of the build cluster with busy cluster

error messages. In this case, LSF reports that all cluster nodes are busy and it will not start any more jobs. These delayed builds begin only when the original jobs finish and the nodes are no longer busy.

In addition to the above original cluster, and at the same time the following investigation and implementation of the MOSIX cluster was underway, another LSF cluster was being assembled for use in another group. This LSF cluster consisted of 37 Sun Ultra 2 workstations, each with dual 400MHz UltraSparc-II CPUs each with 2MB of L2 cache, and 512MB of RAM. These workstations were connected to a Fore ESX-2400 100Mbit switch with 96 ports. We were able to run some test numbers on this cluster, and were able to run a maximum of four users concurrently. While this cluster was not investigated extensively by us, the performance and costs are shown in the figures and charts in relation to the other clusters and nodes.

The Investigation

Due to the problems cited above, we initiated a project to investigate alternative cluster technologies. Given one author's past experience with an earlier version of

MOSIX [Barak et al., 1993], we tried to use the Linux-based MOSIX cluster at Hebrew University [MOSIX] to build our code images.

MOSIX adds load information about the cluster, process migration, and various other clustering features to Linux without adding any new, cluster-specific APIs. This allows us to continue to use the existing compilers and build environments with little to no modifications. (Note that we use a standard GNU tool set on Linux and our other UNIX hosts).

Although MOSIX did an excellent job measuring the load across the nodes, relying on its process migration mechanism to distribute the sub-makes across the cluster nodes failed. MOSIX on Linux redirects many system calls for migrated processes through the network to a stub process that runs on the process' creation node. Since `make(1)` forks children locally and since compilation is system call intensive, few if any migrations actually occurred. When `make(1)` sub-processes were forced to migrate, we observed a huge performance drop.

After discussion with the MOSIX team, we suggested that one way to avoid this performance degradation was to use static placement of the builds instead of migration. This uses the MOSIX information monitoring abilities in order to determine process placement and then uses a local daemon on each cluster node that creates the process on the selected node. This approach avoids the performance degradation of remote system calls by using remote execution and static placement instead of local forking and dynamic migration.

As the MOSIX team worked on the remote executor, EMC started a more in-depth analysis of our existing builds. A snapshot of the build environment and source code was taken and used in all tests to ensure consistent results in our experiments. Our test measured a full, clean build (all previous results were removed from prior runs each time). Like our original cluster, source code and binaries were stored in NFS. It was also noted that large portions of the builds were not parallelized. This problem would have to be tackled later in order to see greater performance gains.

The next step in the investigation was to characterize the performance of single computers and determine a good choice for the nodes of the cluster. After running the build environment on various computing nodes, it was determined that the best hardware platform based on cost, size, network port considerations and

performance was a VA Linux [VA/Linux] 3500. Each VA Linux 3500 is a quad-CPU Intel Xeon computer, running each processor at 500MHz with 512KB of L2 cache on each processor, and 1GB of RAM. Using a single VA Linux 3500, the existing builds for a single user for a single job would take a little over 37 minutes (2233 seconds).

In addition to characterizing the performance of the individual candidate nodes, we profiled the build itself. Using slow-CPU nodes, the majority of a build's elapsed time is spent computing. As we moved to faster CPUs, the builds became increasingly I/O bound. Depending on the particular stage of the build, it could be either I/O bound or CPU bound. Overall, it was determined that the I/O requirements for a single build were huge: reading in the source files, writing out object files, linking binaries, etc. This finding reinforced our understanding of why MOSIX's default migration failed initially: since MOSIX currently does not perform local I/O, it instead sends the I/O back to the node where the process originated. The most recent release of MOSIX allows remote processes to perform I/O to some cluster file systems directly, as explained further in the Future Work section of this paper.

MOSIX Enhancements

The MOSIX team produced a set of tools, called the MExec/MPMake package that built on the kernel load information as described above. This package can be combined with GNU `make(1)` to spawn remote jobs across the cluster by using the load information to determine the best node on which to place the remote job. In this way, processes are distributed across the cluster and are able to do local I/O. The MExec/MPMake package consists of a small server that runs on each node of the cluster. This server acts as a proxy that creates local processes when passed the command name, argument list, etc. from the remote node. EMC sponsored the development of this package and encouraged its release under the GNU Public License (GPL).

A small client sends commands to these servers based on the node with the lowest load in a list supplied to the command. This client only ran on nodes in the MOSIX cluster. An additional, non-MOSIX dependent client was created that can send commands to a specific node in the cluster, so that the remote make can be started on the cluster from computers that are not members of the MOSIX cluster. This client currently runs on Linux, FreeBSD, Solaris, and

Windows. This is useful because users do not have to log directly into the MOSIX cluster; instead, scripts and local makefiles remotely start jobs on the cluster. Process migration is disabled so that processes stay on the nodes where they start.

File System Issues

Consistency is the responsibility of the build environment or the distributed filesystem. EMC addresses this by turning off name, data, and attribute caching in NFS. Otherwise, nodes do not see newly created files or directories, which results in inconsistent or failed builds. While this creates consistency, it also greatly hinders performance, as network traffic is much higher. Users of a cache coherent file system, like GFS [GFS], should be able to run with caching enabled.

The Test MOSIX Cluster

With the arrival of the new MOSIX tools, we built our first test cluster. The test cluster consisted of 8 VA Linux 3500 computers (1GB of RAM, 4-way 500MHz Intel Xeon processors each with 512KB of L2 cache) and one gatekeeper node that is a Compaq 550MHz Intel Pentium III with 128MB of RAM. We used the same backing store as the original LSF cluster, but replaced the original cluster's FDDI ring with a Cisco

6506 100Mbit switch with 96 ports.

The Compaq gatekeeper box is used as a non-computing member of the cluster; all MOSIX build jobs are submitted through this gatekeeper. Since all MOSIX nodes are peers, this is not a necessary step. However, it creates a nice place to keep track of all incoming jobs, run monitoring software, or control access to the cluster without wasting an expensive machine. Note that if this node is removed, jobs can be sent directly to the computing members of the cluster.

On this test cluster, the total time for a single user to build alone on an idle cluster was about 9 minutes (537 seconds).

The next test was to have multiple users running jobs at the same time. Running with six different users concurrently, the time for each user to complete their build was 26 minutes (1560 seconds). Note that all users were slowed down equally, *i.e.* they all shared the load of the cluster equally.

The Production MOSIX Cluster

Once we proved that the test cluster worked well, we added 16 additional nodes, bringing the total up to 24 VA Linux 3500 computers (now shipping with 550MHz CPUs) (see Figure 2).

Figure 2: The Mosix/Linux Cluster

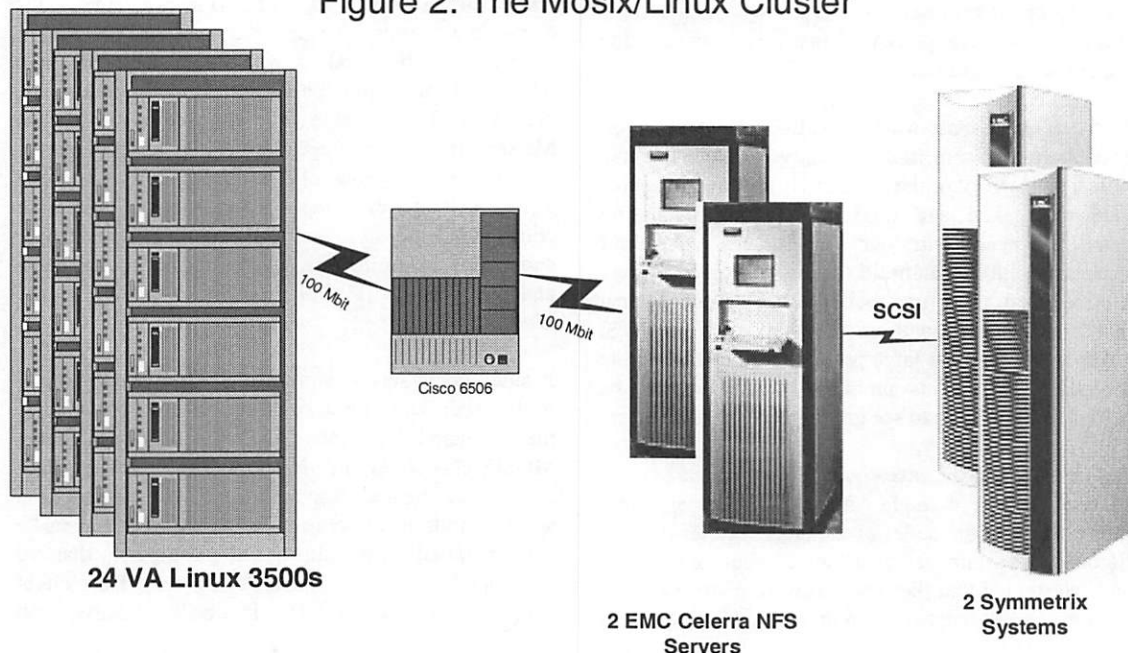
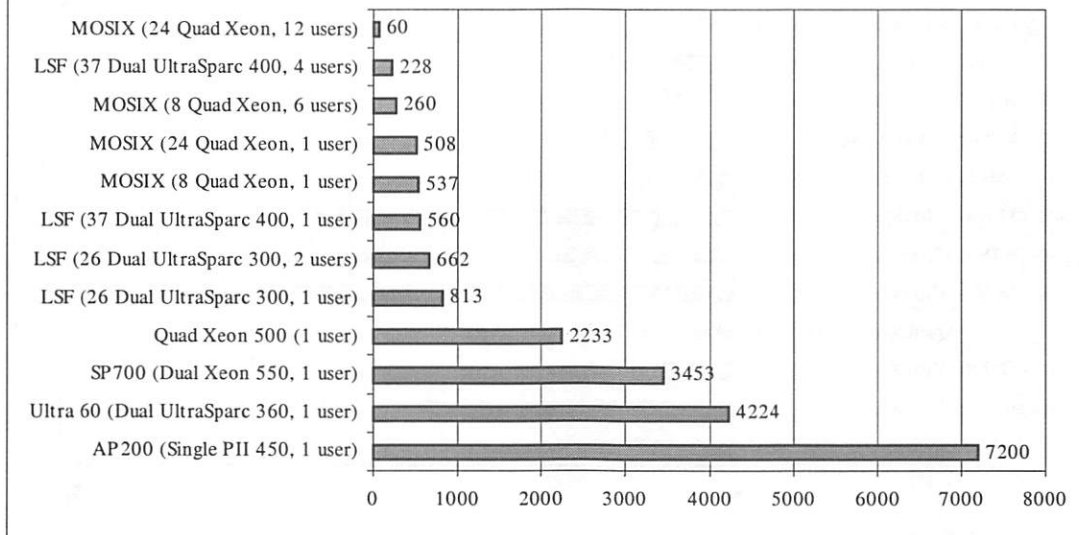


Figure 3: Average seconds per build per user



On the next test, all 24 nodes were used. An interesting effect, as shown in Figure 3, was that the build time remained mostly constant for a single user, yielding a slightly improved build time of 8.5 minutes (508 seconds). Further investigation using a network analyzer revealed that the 100Mbit connection from the switch to the NFS data mover was completely overwhelmed by the compute nodes. We set up a network analyzer to monitor traffic on the switch. The switch showed that once we started the build process, the link from the switch to the data mover would become 100% utilized, while the links to the individual nodes would be at most 5% to 10% utilized. Upgrading the network is the next step to overcoming this limitation.

The most noticeable improvement occurred when multiple users would build simultaneously. With 12 users building concurrently on different NFS data movers, the time for each user to complete a build increased from 8.5 minutes (508 seconds) to about 12 minutes (715 seconds). Although this is about 40% slower than the time spent building on an idle cluster, it is still faster than a single user running a build on the original LSF cluster when idle. If two or more users share an NFS data mover, those users do see significant decreases in performance as they are contending for the same physical device.

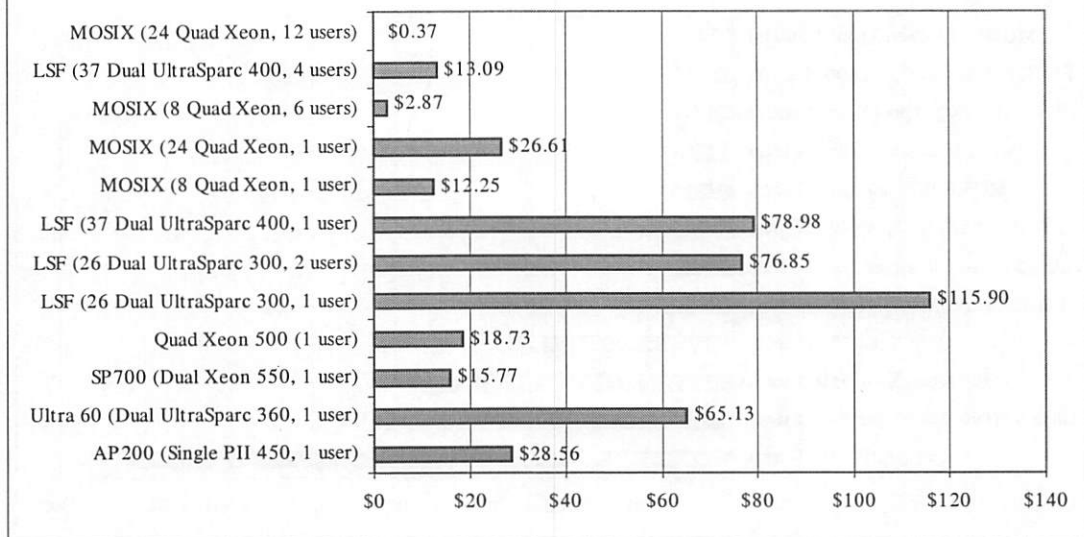
Performance Analysis

In Figure 3, we detail the average build time in seconds per user for the various platforms tested. The numbers were achieved by taking the time in seconds, from start to finish, of the user's build jobs and then dividing by the number of users building concurrently to get the average build time per user.

As Figure 3 shows, we were able to gain approximately 1100% improvement in performance when 12 users are using the Production MOSIX Cluster as compared to the two users using the LSF Cluster. Note that the original LSF cluster allowed a maximum of two users at once. This was a major restriction since we potentially have upwards of 20-25 developers trying to build concurrently.

One of the bottlenecks we observed (as shown before) was the network connection to the NFS data movers being saturated by the compute nodes in the cluster. Preliminary testing on one NFS data mover shows that upgrading the link between the switch and the data mover to Gigabit Ethernet reduces build times by approximately 23%. Upgrading this link and the data mover to its newest release reduces build times by approximately 42%. Our best times were measured with the Gigabit Ethernet, upgraded data movers, and with a striped disk file system behind the NFS data

Figure 4: Average cost per build per user (normalized to 1440 builds daily)



mover. This configuration reduces our current build times by 51%, making the cluster more than twice as fast.

Cost Analysis

For the original LSF cluster, total cost is approximately \$660,000, with software costs of \$61,141.25 up front and an annual cost of \$16,835 for support licenses for a two year life span, as shown in Figure 4. The Production MOSIX cluster cost is approximately \$390,000, with no recurring software costs, for a two year life span, also shown in Figure 4. In addition, having the full source code of the system and tools enables us to debug and fix problems as they arise. Detailed cost information is shown in Appendix A.

Overall, cluster costs were reduced by 41% (about \$270,000) while performance was increased at least eleven fold. In Figure 4, we use these numbers to compute the cost per build per user (assuming a two year life span for each cluster). In all cases, the chart assumes that builds run 24 hours a day. The numbers are all normalized to the equivalent of building 1440 builds per day (the amount that the fastest cluster – the Production MOSIX cluster – could sustain). Stated another way, the numbers show what the cost per build

is if you could linearly scale each solution to achieve 1440 builds per day.

Administration

The Production MOSIX Cluster administration consists of monitoring the cluster for availability. Once the cluster was turned on for general use, the cluster has needed little to no attention or intervention other than two hardware memory failures. Users can submit jobs directly from Linux, FreeBSD, Windows, or Solaris clients.

By comparison, the LSF Cluster administration consists of maintaining a license server for the LSF cluster management software and monitoring the cluster for availability. Only Solaris clients can submit jobs directly to the cluster, and each Solaris client wishing to submit jobs to the cluster must be pre-configured by the cluster administrator as a client to the cluster.

Future Work

We are currently working with the MOSIX team to explore several enhancements to the cluster. One obvious shortcoming is the lack of support for a cache-coherent, distributed file system like GFS. Working

with EMC, the MOSIX team has added this support, enabling a process to issue its I/O system calls locally, even after migration. This should allow the cluster to use its normal dynamic process migration for a larger group of I/O intensive tasks, including our distributed builds. EMC's efforts will focus on tuning the rest of the components of the system: the network will be upgraded to Gigabit Ethernet and users' directories will be striped across multiple NFS data movers to exploit EMC's high-end storage systems.

Related Work

Process migration has been an active area of research since the early 1980s, but has had limited success in the commercial world. In early systems, like the original MOSIX implementation, migration was added to existing systems by restructuring the internals of the base operating system. This approach produced a great deal of transparency: user processes could migrate freely without any requirement to link against special libraries or use special system calls. A significant drawback was the difficulty of maintaining the migration code in the continually changing host operating system.

Sprite [Douglass and Ousterhout, 1987], developed at UC Berkeley, simplified this process by introducing the concept of the home node as is used in the current MOSIX implementation. In this scheme, migrated process redirect some of their system calls back to their creation node. This retained the transparency of migration for user processes, but also had a performance impact for certain types of processes.

In the same era, work started on user-level process migration. This approach traded transparency and performance for portability. For example, little to no changes were required to port Condor [Litzkow, 1987] to a new host platform. Condor had a strong influence on the development of Utopia [Zhou et al., 1994], the academic precursor of LSF, and Loadleveler from IBM.

The rare commercial implementations of process migration include Locus Computing's migration support in OSF1/AD for the Intel Paragon [Zajcew et al., 1993] and Platform Computing's LSF.

For a comprehensive survey of process migration, see Milojicic, et al [Milojicic et al., 2000]. In this survey, many of the seminal migration papers are presented as

a collection, including the early MOSIX, Condor, and Sprite papers.

Conclusion

Our project demonstrates that it is possible to create high performance, distributed build environments from commodity hardware and open source software, such as Linux and MOSIX. In addition, this project demonstrates that collaboration between the open source community, an industrial systems group, and our system administrators works well. Together, we produced a system with an order of magnitude performance improvement at less cost than our original cluster. In the future, as software becomes more complex and requires more CPU processing power, the cost benefits will become both more apparent and more important.

Acknowledgements

We would like to thank Amnon Barak and Amnon Shiloh from the MOSIX team for putting together a great cluster package for Linux clusters. We would also like to acknowledge the efforts of Kathie Graceffa, who lead the EMC MOSIX cluster project from the system administration side, Varina Hammond and Clem Cole for their comments and revisions of this paper.

References

- Barak, A., Guday, S., and Wheeler, R. (1993) The MOSIX Distributed Operating System. *Lecture Notes in Computer Science, Vol. 672, Springer-Verlag*.
- Douglass, F. and Ousterhout, J. (September 1987). Process Migration in the Sprite Operating System. *Proceedings of the Seventh Conference on Distributed Computing Systems*, pages 18-25.
- EMC² Corporate Home Page: <http://www.emc.com>
- Global File System Home Page:
<http://www.globalfilesystem.org>
- Litzkow, M. (June 1987). Remote UNIX – Turning Idle Workstations into Cycle Servers. *Proceedings of the Summer USENIX Conference*, pages 381-384.

Milojicic, D., Douglass, F., Paindaveine, Y., Wheeler, R., Zhou, S. (to appear in 2000). Process Migration Survey. *ACM Computing Surveys*.

Milojicic, D., Douglass, F., and Wheeler, R. (February 1999). Mobility: Processes, Computers and Agents. *Addison-Wesley Longman and ACM Press*.

MOSIX Project Home Page: <http://www.mosix.org>

Platform Computing Corporate Home Page:
<http://www.platform.com>

Popek, G. and Walker, B. (1985): The Locus Distributed System Architecture. *MIT Press*.

VA/LINUX Corporate Home Page:
<http://www.valinux.com>

Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., and Netterwala, D. (January 1993). An OSF/1 UNIX for Massively Parallel Multicomputers. *Proceedings of the Winter USENIX Conference*, pages 449–468.

Zhou, S., Zheng, X., Wang, J., and Delisle, P. (December 1994). Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software-Practice and Experience*.

Appendix A

Table 1 shows the cost of the individual cluster nodes, based on the full manufacturer's list price as of February 1, 2000. Note that any possible discount was not factored into the calculations.

Table 1: Hardware prices	
VA Linux 3500 Quad Xeon - 1GB/550Mhz	\$14,215.00
Compaq AP200 PII - 128MB/450Mhz	\$2,085.00
Compaq SP700 Dual Xeon - 512MB/550Mhz	\$5,004.00
Sun Dual Ultra 2 - 512MB/300Mhz/FDDI	\$20,870.00
Sun Dual Ultra 2 - 512MB/400Mhz	\$21,090.00
Sun Dual Ultra 60 - 512MB/360Mhz	\$13,815.00
Cisco 6506 + 96 100BaseT ports	\$46,975.00
Fore ESX-2400 + 96 100BaseT ports	\$40,780.00

Table 2 shows the total costs of each of the clusters, including network and software costs, but does not include the cost of the storage (since the storage costs are constant and can be shared across clusters).

Table 2: Cluster prices (2 year life)	
LSF (26 Dual UltraSparc 300, FDDI)	\$663,601.25
LSF (37 Dual UltraSparc 400, 100Mbit)	\$953,036.25
MOSIX (8 Quad Xeon, 100Mbit)	\$160,695.00
MOSIX (24 Quad Xeon, 100Mbit)	\$390,220.00

Webmin

A web-based system administration tool for Unix

Jamie Cameron (jcameron@calderasystems.com)
Caldera Systems

Abstract

This paper describes the design and implementation of the Unix administration tool Webmin, available from <http://www.webmin.com/webmin/>. Webmin allows moderately experienced users to manage their Unix system through a web browser interface, instead of editing configuration files directly. The most recent version supports Apache, Squid, BIND, Samba and many other servers and services. It supports multiple operating systems and distributions, different languages, multiple users each with different levels of access, and SSL encryption.

The first part of the paper explains why Webmin was developed and the initial design goals, and compares the design to other similar tools such as Linuxconf. Subsequent sections cover the design and implementation of the detailed multi-user security model, the implementation of Webmin itself, how support for multiple operating systems is handled and how internationalization works. Finally, two Webmin modules are discussed in more detail and various problems explained before the conclusion.

1 Introduction

For the inexperienced user, Unix system administration can be daunting. Almost all services have configuration files that must be edited manually and often have complex formats. While these files are usually well documented in *man* pages, it is often unclear exactly how different sections and directives fit together. Furthermore, a single misspelling or missing punctuation character can ruin an entire configuration file.

To make matters worse, similar services have configuration files with different structures and locations on different kinds of Unix systems. For example, Solaris stores NFS exports in */etc/dfs/dfstab*, while Linux, FreeBSD and HP-UX use */etc/exports* – and all four use different formats. For an inexperienced system administrator in charge of several different

types of systems, these inconsistencies can make life difficult.

My aim in writing a system administration tool was to solve both these problems. It needed to provide a friendly user interface with basic error checking, and consistency across the many different Unix systems and distributions. Additional goals were completeness (all reasonable options for the service being configured should be settable) and non-destructiveness (comments and unknown options should be unharmed).

This paper describes the design, implementation and future of my system administration tool Webmin. Section 2 explains the design of the system, section 3 the implementation, section 4 covers two modules in detail, section 5 discusses problems encountered and finally section 6 concludes the paper.

2 Design

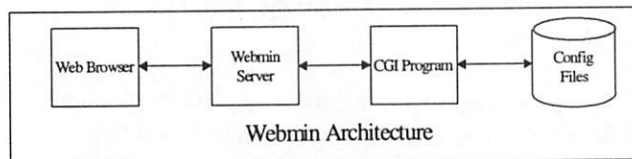
2.1 System architecture

I decided early on that the easiest and best administration user interface was one accessible through a web browser. A web-based interface is platform independent, easy to develop and accessible locally or over the network. I then began writing CGI programs in perl to be run under an Apache webserver, but eventually developed my own webserver written entirely in perl to remove the dependency on Apache, which may not be available on users' systems and dislikes running as root.

Perl was the natural choice of implementation language for this project, as it has strong string processing facilities for reading and updating configuration files, and is available on every Unix platform that I might want my administration tool to run on. Avoiding the need for compilation was also an important goal, as some Unix variants do not even ship with a compiler and there are many ways compilation can fail due to the lack of a key library. With perl, all

these problems have effectively been already solved by the perl installation process.

While the web / perl / CGI architecture is simple and easy to develop, it is not perfect. Because almost every Webmin screen is generated by a CGI program, the web server must fork a new process for each page, which can be slow on underpowered or heavily loaded systems. This has been resolved in the most recent versions by having the web server execute CGI programs in-process, in a similar way to mod_perl. Response time can also be a problem when managing distant servers, as every form submission must be processed by the web server. The user interface capabilities of HTML are also inferior to what is possible with a real user interface toolkit such as Qt or Swing, although they can be improved somewhat through the use of Javascript.



2.2 Alternative architectures and tools

I considered two other possible architectures before deciding on the perl / CGI design :

- **A standard X11 application, written in C or C++ and using Motif or Qt**

This would have some advantages from a user-interface point of view, as X applications have far more controls and inputs available than web applications. However, C is not a good language for writing string and file manipulation code in and remote access would only be possible if the user was running an X server. Because remote access was a major design goal, a browser-based solution was preferable.

- **A Java applet client and server**

This option solves the remote access problem (because applets run in a web browser), while still allowing complex user interfaces to be developed. Unfortunately, at the time Webmin was designed Java applets were still rather unreliable, especially large and complex programs. In addition, Java's string manipulation capabilities are not much better than C and far inferior to Perl.

When I started development in 1997, Webmin was not the only administration tool for Unix system. Others

available were :

- **Linuxconf**

While today Linuxconf is a very impressive tool with many of the same functions as Webmin, at the time its development had only just begun. Its internal design is different to Webmin in that instead of writing to config files directly, it stores an internal list of changes, which is only written out when the *Activate Changes* button is clicked. It has a modular architecture like Webmin, though the modules are shared libraries written in C++ rather than sets of CGI programs, and it supports multiple languages for the user interface and help screens.

Linuxconf's method of batching config file changes has some advantages, such as the ability to have multiple related changes applied simultaneously. The biggest disadvantage is the possibility that changes made manually or by other tools could be overwritten unexpectedly if Linuxconf's modifications have not been applied for some time.

While it supports text, GTK and browser-based user interfaces, most of the effort seems to have gone into the GTK interface. This makes Linuxconf excellent for local administration, but not as good for remote administration through a web browser. Other problems are the lack of support for non-Linux systems and the inability to handle multiple users with different levels of permissions.

- **Redhat Control Panel**

This is a collection of small programs shipped with Redhat Linux for configuring users, printers, networking and a few other services. While it is good for those few services, it can only run as an X application and only supports Redhat Linux config files.

- **SAM**

This is a X/Motif administration application shipped with the HP/UX operating system. Like the Redhat control panel, it does not support remote access and only allows the configuration of a few services such as NIS, users accounts and networking.

2.3 Modules

Almost all of Webmin's functions are divided into modules, each of which is a mostly independent set of CGI programs responsible for managing some Unix feature or service. The programs for each module are

stored in a separate subdirectory below the base Webmin directory (the webserver document root), and thus each module is accessed by a URL like `http://server:10000/module/`. Every module has some information associated with it, such as a human-readable description, the operating systems it supports, and the other modules that it depends upon.

When a user first logs in to the server, he will be presented with a list of all the installed modules to which he has access, with each module displayed as an icon from its directory. Each module is displayed in one of five categories (Webmin, System, Servers, Hardware and Others), with the module itself determining which category it is in. This layout was adopted after the original method of showing all modules on one page became too large and cluttered.

The module system makes the distribution and addition of new third-party modules simple. Third-party modules can be distributed as Unix TAR files (normally with a `.wbm` extension) for installation into existing Webmin servers. Because each is fully self-contained, the installation process consists of nothing more than untarring the module file and adding it to the access list of the current user.

At the time of writing Webmin has 38 standard modules, capable of configuring the Apache webserver, Unix users and groups, Samba, Squid, Sendmail, NFS exports, disk partitions and more. There are also 17 third-party modules for services such as IPchains, Qmail, NetSaint and others.

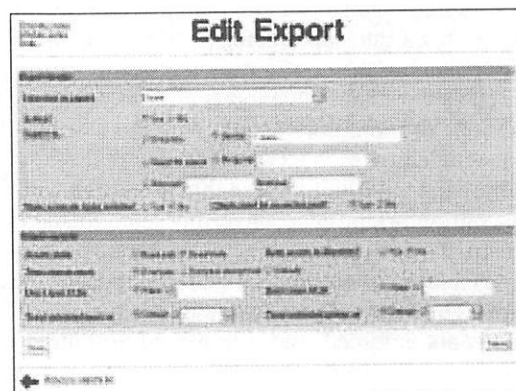
2.4 User interface design

Because Webmin uses a web / CGI architecture, all user interfaces are simply HTML pages and forms. The major goals in designing the Webmin user interface were :

- **Consistency**
I wanted all screens to have a consistent look and feel, so that novice users could easily find their way around. This meant using a standard color scheme, title layout, footer and common interface elements, such as tables of icons.
- **Simplicity**
To make screens easy to use (and to avoid the problem with large forms in some browsers), I wanted to limit the amount of information and form inputs displayed on each screen. To achieve this, many functions are broken down into multiple screens although it would be possible to put all the

information on one page.

- **Compatibility**
Because there are so many web browser types and versions in use, I chose to use only the lowest level of features possible. This meant no frames, DHTML, Javascript or Java unless the there was no alternative (for example, the File Manager feature is written in Java because it could not be done any other way), or for adding non-vital features (such as the optional file selection dialog in some screens).



2.5 Security

Because Webmin runs through a web server, the first level of security is a standard HTTP login prompt displayed when the user attempts to access the server. To prevent brute-force attacks, Webmin can be configured to delay an increasing amount of time before responding to each incorrect login. This will not defend against sniffing the network for the password of a valid user, which is why Webmin is also capable of running in SSL mode if the OpenSSL library has been installed on the system it is running on.

A key feature of Webmin is support for multiple users, each with different levels of access. The initial design only supported granting each user either total access to a module, or no access at all. This turned out to be inadequate however, as many modules could be used in ways to gain root access and thus total control of the system – for example, a Webmin user granted access to all cron jobs could just create a job run as root and thus take over the system.

The solution to this problem was the creation of an additional more fine-grained level of security. This allowed the granting to users only certain functions of each module, rather than total control. For example, it is possible to give a Webmin user the right to edit cron jobs only for selected Unix users, or the right only to

manage certain Apache virtual servers. This feature can be very useful if a master administrator wants to delegate some tasks to other admins, without giving them access to the entire system.

3 Implementation

3.1 Introduction

Webmin is implemented as a large number of perl CGI programs, arranged into subdirectories called modules. Each module handles the configuration of some Unix service, such as Apache, NFS exports or cron. Each module has one or more libraries of common functions, included by each CGI program with the *require* command. Typically, these functions deal with the actual configuration files, converting them to and from data structures used by the actual programs. That way, there is a layer of abstraction between most of the CGI programs and the system – although this abstraction is not enforced, and is bypassed in some cases.

By abstracting the user interface away from the actual configuration files, not only is much repeated code avoided, but also the possibility is opened up for modules to call each other. For example, the module *fdisk* (Partitions on Local Disks) has a library *fdisk-lib.pl* that contains functions for discovering the disks and partitions on Linux systems. These functions are called by the *raid* (Linux RAID) module to find disks available for inclusion into a RAID array, by the *lilo* (Linux Bootup Configuration) module to display bootable partitions, and by the *mount* (Disks and Filesystems) module to display mountable partitions.

Webmin has its own web server called *miniserv* that comes as part of the distribution. Unlike a general-purpose webserver such as Apache, it has very few features and only supports the retrieval of files and the execution of CGI programs. Because it is written in Perl like the rest of the Webmin programs, it has the ability to run CGIs in-process without the need to *fork* and *exec* a new Perl interpreter, in a similar manner to the *mod_perl* Apache module. This provides a substantial speed increase on slow or low memory systems.

3.2 Common functions

In addition to the function library in each module, there is a file named *web-lib.pl* in the top-level

Webmin directory that contains functions used by all modules. Every CGI program must *require* this library, either directly or by via its module library. In addition, every CGI program must call the function *init_config* to read in the module configuration file, perform security checks and set several global variables. Typically, this function is called by the module's function library just after including *web-lib.pl*.

web-lib.pl includes many different functions, which can be roughly broken down into the following categories :

- Standard user interface functions for generating headers and footer, icon tables, user and file dialogs and so on. These help provide a consistent look across all Webmin screens.
- File manipulation functions for inserting, replacing and deleting lines in configuration files.
- Network functions, for downloading files via FTP and HTTP.
- Functions for making 'foreign' function calls. These are calls from CGI programs in one module to functions in another module's library, done in such a way that the called function has its environment set up just as if it was being called by one of its own CGI programs.
- Webmin-specific functions for getting information about other modules, access control, the Webmin version and so on.
- Assorted convenience functions, including CGI functions for reading form inputs into perl variables.

3.3 Platform Independence

In order for Webmin to work on the many different Unix variants and Linux distributions, it needs to know exactly which operating system and version the user is running. This information is determined at install time, either by asking the user or automatic detection from the */etc/issue* file or the output of *uname*. Once the operating system is known, the appropriate configuration for the OS is selected from each module and used to find and parse system config files. Activate configuration files are stored in the */etc/webmin* directory, with each module having its own subdirectory for its configuration file and any other temporary files that it might create.

For example, the location of the Apache *httpd.conf* file differs between every Linux distribution, and even between different versions of the same distribution. Webmin can deal with this though, as it knows where *httpd.conf* is located on all the different operating systems and distributions that it supports. Possible alternatives would be to ask the user where every config file is (not very user friendly) or to find the config files automatically by searching the entire filesystem (not always possible, as many config files do not exist initially).

Because some operating system behavior is too complex to encode in the simple Webmin config files, some modules have separate Perl libraries for each OS. For example, in the Disk Quotas module each OS library implements the same set of functions, but in a different way to handle the different ways quotas work on each operating system.

3.4 Internationalization

In order to support different languages, all the text strings from most Webmin modules are no longer hard-coded into the programs, but are instead stored in separate language files. Each module has one file per supported language, the one to use being selected by the user's current choice of language. When a program needs to display some text or message, it uses a message code that is looked up in the appropriate language file and converted to the actual text in the correct language.

Because some words like *Save*, *Create* and *Default* are used by many modules, there is a master set of language files that contain these messages for the use of all modules. These master files also store messages used by the main Webmin menu and some programs that are used by all modules.

The online help system also supports multiple languages, with each help page stored in all the supported languages. When a help page is requested, the file for the current language is read and displayed. A similar selection process is also used for displaying the module configuration page in the correct language.

With all translation in Webmin, if a message or page is not available in the user's currently selected language then the default language (currently English) will be used instead. This means that partial translations can be contributed and are still useful, and that the addition of new messages will not require the immediate update of all language files.

3.5 Security Implementation

Because Webmin runs through a web server, the first level of security is simply HTTP authentication enforced by the web server using usernames and passwords from a standard Apache-style users file. Because the server also enforces checks against brute-force attacks and because the authentication protocol is relatively simple, there is little chance of an attacker breaching this level of security.

The second security level is enforced by module CGI programs themselves, all of which check the HTTP username against the list of allowed users for the module. This checking is done by the common *init_config* function which every program directly or indirectly calls. Because of this, any program that does not check whether the current user is allowed to access it could theoretically be a security hole. A better alternative would be to have this level enforced by the webserver, although that would complicate running Webmin under other web servers such as Apache, as any change to the list of modules a user has access to would require changing the Apache configuration as well.

The third security level of fine-grained module access control is also enforced by the CGI programs, but the complexity and variability of this access control means that there is no common function that the programs can call. Instead, each program checks a list of actions allowed by the current user and displays an error message if the action the user requested is unavailable. This means that the potential for accidentally creating a security hole is much greater, but I see no other alternative.

The Webmin webserver can also use the *OpenSSL* and *Net::SSL* libraries to encrypt communication between the server and browser with SSL. Thanks to these libraries, the implementation of SSL is relatively simple – the only difficulty is the provision of an SSL certificate, which normally must be come from a trusted source like Verisign, and associated with the hostname of the server. Because Webmin's certificate is not valid, there is no defense against man in the middle attacks that trick the user into thinking he is accessing his Webmin server when he really is not.

4 Module Discussions

4.1 Apache Webserver

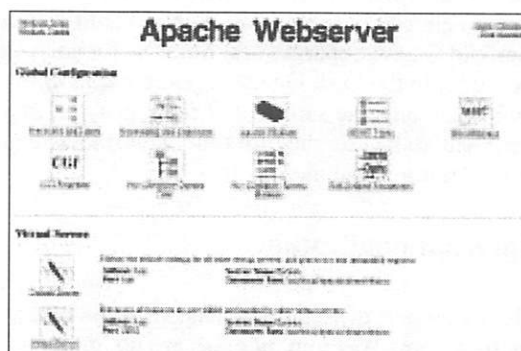
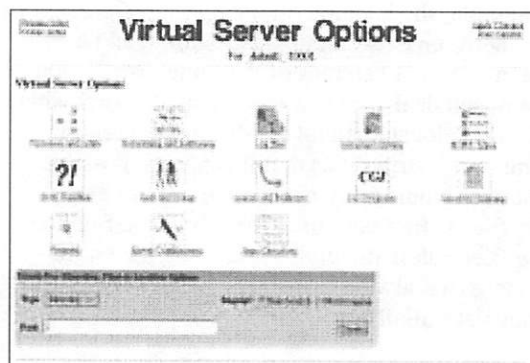
The Apache Webserver module is one of the most complex, due to the massive number of configuration file directives supported by Apache. My objective was to support all common Apache versions, and as many different directives as possible. This was made more complex by the large number of Apache modules, an unknown combination of which could be compiled into each user's Apache installation, and each of which could add several new directives.

Fortunately, it is possible to query an Apache installation for version number, compiled in modules and dynamically loaded modules. This information combined with the Webmin module's built-in knowledge of the availability of each directive in each Apache release makes it possible to edit the Apache config files correctly. However, keeping up with the changes in each Apache release can be difficult.

Once the supported modules are known, a Perl library for each Apache module is read in, each of which contains a function for editing and updating each module directive. Each directive in each module is assigned a category (such as *Log Files* or *Access Control*), so that when the user visits the *Log Files* submenu inputs for all the directives in all modules related to that category are displayed. This categorization system allows the large number of directives to be broken down logically instead of all being displayed on one huge page.

Because Apache supports subsections such as `<VirtualHost>` or `<Directory>`, the user interface is further categorized to allow the editing of directives in those sections, which are also broken down into categories. This way even a large Apache configuration with many virtual servers can be easily managed, and all supported directives can be edited in all sections.

Because there are some Apache directives (such as those in the `mod_rewrite` module) that are too complex for Webmin to manage, the module also provides support for manually editing parts of the config files. Comments and unsupported directives such as these are not effected by the other pages in the module.



4.2 Users and Groups

The Users and Groups module is designed to allow the creation, modification and deletion of Unix users and groups by directly editing the `/etc/passwd`, `/etc/group` and `/etc/shadow` files. This is complicated slightly by the difference in format and existence of those files on different operating systems – for example, some Linux distributions do not have an `/etc/shadow` file at all, while some BSD-derived systems use the file `/etc/master.passwd` as their main source of user account information.

This functionality already exists in many other administration tools however. What makes Webmin unique in this area is the ability to update other parts of the system when a user is created or modified. For example, you can configure the Samba module to have a user added to Samba's encrypted password file whenever a Unix user is added, something that normally must be done manually. Similarly, the administrator can setup default quotas to be assigned when new users are created.

5 Problems

After several years of development and use, I have identified several design and implementation problems in Webmin :

- **Second and third level security**
This is still dependent on checking by individual CGI programs, and thus security breaching are more likely than they should be. While the second level is relatively secure and can be improved, there is no easy way to improve the fine-grained third level of access control. Only careful coding and close examination of the code by others can help solve this problem.
- **Coding style**
The development of Webmin began when I was unfamiliar with Perl 5 features such as packages and modules, and unfortunately the initial coding style has continued through to this day. A good implementation would have made each module a separate Perl module, allowing modules to call each other using standard Perl syntax instead of the ugly *foreign_* functions. The best long-term solution is to recode all the Webmin modules as proper Perl modules, so that function calls between them can be made with Perl's *module::function* syntax.
- **Internationalization**
Because multi-language support was not part of the original design, several older modules still have strings hard-coded into their programs. Internationalization of these modules requires nothing more than time and tedious work, as there is no reliable way this process could be automated.
- **Keeping up with new releases**
Because new releases of servers such as Samba, Apache and Squid often have new configuration directives or change the meanings of existing ones, Webmin must keep up with the latest version of these programs. In addition, new releases of operating systems and Linux distributions which add new features and change the locations of config files much also be kept track of. This is an unavoidable task, but relatively easy if I download each new release of the supported servers and install each new version of operating systems and distributions to which Webmin has been ported. Fortunately, programs such as *VMware* make the testing of new PC-based operating systems relatively easy.

- **Documentation and help**

Most modules still lack online help, and there is no overall manual or instructions on how to use Webmin. While most screens are relatively self-explanatory to experienced system administrators, beginners would benefit from a simple set of instructions on what DNS domains are and how to set them up, how to create Unix accounts and so on.

6 Conclusion

From its initial releases that contained only a few modules, Webmin has met its design goals and developed to support many commonly used Unix services, becoming a useful and powerful administration tool. This paper has covered the design and implementation of Webmin, and should provide information for others planning to develop similar tools or contribute to Webmin.

7 Availability

Webmin is distributed under a BSD-like license, and thus is freely available. It can be downloaded from <http://www.webmin.com/webmin/> in *.tar.gz* , *RPM* and Solaris package format. Several third-party modules contributed by others can also be downloaded from the same URL.

Porting the SGI XFS File System to Linux

Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan, Ken McDonell, Ted Kline,
Brian Gaffey, Rajagopal Ananthanarayanan

SGI

Abstract

The limitations of traditional Linux file systems are becoming evident as new application demands for Linux file systems arise. SGI has ported the XFS file system to the Linux operating system to address these constraints. This paper describes the major technical areas that were addressed in this port, specifically regarding the file system interface to the operating system, buffer caching in XFS, and volume management layers. In addition, this paper describes some of the legal issues surrounding the porting of the XFS file system, and the encumbrance review process that SGI performed.

1. Introduction

In the early 1990s, SGI realized its existing file system, EFS (Extent File System) would be inadequate to support the new application demands arising from the increased disk capacity, bandwidth, and parallelism available on its systems. Applications in film and video, supercomputing, and huge databases all required performance and capacities beyond what EFS, with a design similar to the Berkeley Fast File System, could provide. EFS limitations were similar to those found recently in Linux file systems: small file system sizes (8 gigabytes), small file sizes (2 gigabytes), statically allocated metadata, and slow recovery times using fsck.

To address these issues in EFS, in 1994 SGI released an advanced, journaled file system on IRIX¹; this file system was called XFS[1]. Since that time, XFS has proven itself in production as a fast, highly scalable file system suitable for computer systems ranging from the desktop to supercomputers.

To help address these same issues in Linux as well as to demonstrate commitment to the open source community, SGI has made XFS technology available as Open Source XFS², an open source journaling file system.

1. SGI's System-V-derived version of UNIX

2. <http://oss.sgi.com/projects/xfs>

Open Source XFS is available as free software for Linux, licensed with the GNU General Public License (GPL).

As part of our port of XFS, we have made two major additions to Linux. The first is *linvfs*, which is a porting layer we created to map the Linux VFS to the VFS layer in IRIX. The second is *pagebuf*, a cache and I/O layer which provides most of the advantages of the cache layer in IRIX. These additions to Linux are described in this paper.

2. The File System Interface

The XFS file system on IRIX was designed and implemented to the vnode/VFS interface[2]. In addition, the IRIX XFS implementation was augmented to include layered file systems using structures called "behaviors". Behaviors are used primarily for CXFS, which is a clustered version of XFS. CXFS is also being ported to Linux. Much of XFS contains references to vnode and behavior interfaces.

On Linux, the file system interface occurs at 2 major levels: the file and inode. The file has operations such as `open()` and `read()` while the inode has operations such as `lookup()` and `create()`. On IRIX, these are all at one level, vnode operations. This can be seen in figure 1, which shows the mapping of Linux file system operations to vnode operations such as XFS uses.

In order to ease the port to Linux and maintain the structure of XFS we created a Linux VFS to IRIX VFS mapping layer (*linvfs*).

2.1 The VFS Mapping Layer (*linvfs*)

For the most part, the XFS port to Linux maintained the vnode/VFS and behavior interfaces[3]. Translation from file/inodes in Linux to vnodes/behaviors in XFS is performed through the *linvfs* layer. The *linvfs* layer maps all of the file and inode operations to vnode operations.

Figure 1 shows the mapping of Linux VFS to IRIX VFS. In this figure, the Linux file system interface is shown above the dotted line. The bottom half of the figure shows the file system dependent code, which resides below the inode.

The two major levels of the Linux file system interface, file and inode, are shown in the figure. Each of these levels has a set of operations associated with it. The dirent level, also shown in the figure, has operations as well, but XFS and most other file systems do not provide file system specific calls.

The linvfs layer is invoked through the file and inode operations. This layer then locates the vnode and implements the VFS/vnode interface calls using the semantics that XFS expects.

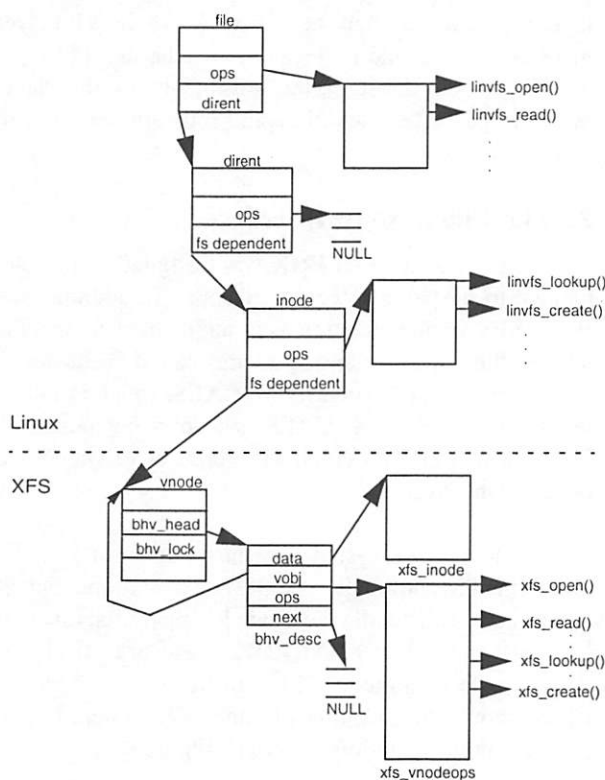


Figure 1: Mapping a Linux VFS Operation to an IRIX VFS Operation.

Linux has three separate types of file and inode operations: directory, symlink, and regular file. This helps split up the functionality and semantics. If the file system does not provide a specific operation, a default action is taken.

Information on the XFS Linux I/O path itself is provided in section 3.8, File I/O.

The linvfs layer is a porting device to get XFS to work in Linux. linvfs allows other VFS/vnode-based file systems to be ported to Linux.

2.2 linvfs Operation Examples

The following examples show how three operations are performed in the linvfs layer.

Example 1: The lookup operation

The lookup operation is performed to convert a file name into an inode. It makes sense to do a lookup only in a directory, so the symlink and regular file operation tables have no operation for lookup. The directory operation for XFS is `linvfs_lookup`:

```
struct dentry * linvfs_lookup(struct inode
*dir, struct dentry *dentry)
{
```

First, get the vnode from the Linux inode.

```
vp = LINVFS_GET_VP(dir);
```

Now, initialize vnode interface structures and pointers from the Linux values:

```
/*
 * Initialize a pathname_t to pass down.
 */
bzero(pnp, sizeof(pathname_t));
pnp->pn_complen = dentry->d_name.len;
pnp->pn_hash = dentry->d_name.hash;
pnp->pn_path = (char *)dentry->d_name.name;
```

```
cvp = NULL;
```

```
VOP_LOOKUP(vp, (char *)dentry->d_name.name,
&cvp, pnp, 0, NULL, &cred, error);
```

If the lookup succeeds, `linvfs_lookup` gets the inode number from the vnode. The inode number is needed to get a new Linux inode. XFS was modified to set this new field, `v_nodeid`, for Linux.

```
if (!error) {
    ASSERT(cvp);
    ino = cvp->v_nodeid;
    ASSERT(ino);
    ip = iget(dir->i_sb, ino);
    if (!ip) {
        VN_RELE(cvp);
        return ERR_PTR(-EACCES);
    }
}
```

In all cases of `linvfs_lookup`, an entry is added to the Linux dcache.

```
/* Negative entry goes in if ip is NULL */
d_add(dentry, ip);
```

If the lookup fails, `ip` will be `NULL` and a negative cache entry is added, which is an entry that will return an indication of not found on subsequent lookups. Subsequent lookups of the same pathname will not reach `linvfs_lookup` since the `dentry` will already be initialized. If the pathname is modified by an operation such as `create`, `rename`, `remove`, or `rmdir`, the `dcache` is modified to remove the `dentry` for this pathname.

Most of the functionality of the lookup operation occurs below `VOP_LOOKUP` and `iget()`. `VOP_LOOKUP` may read the disk to search the directory, allocate an `xfs_inode`, and more.

`iget()` is a Linux routine that eventually calls the file system specific `read_inode()` super operation. This routine required a new VOP, `VOP_GET_VNODE`, which simply returns the already allocated `vnode` so the `inode` can be initialized and point to the `vnode`. The `vnode` is actually allocated and initialized by `VOP_LOOKUP`.

The `VOP_LOOKUP` functionality is much broader than what is expected in the Linux lookup operation. For now, the XFS port keeps the `VOP_LOOKUP` functionality and just requires the file system to provide a new `VOP_GET_VNODE` interface where the `vnode` can be found and linked to the `inode`. In the future, this could be split such that the `xfs_iget()` code could be moved into `linvfs_read_inode()`.

Example 2: The `linvfs_open` operation

An example of a file system operation is `open()`. For XFS, the operation is currently `linvfs_open()` for files and directories and is as follows:

```
static int linvfs_open(
    struct inode *inode,
    struct file *filp)
{
    vnode_t *vp = LINVFS_GET_VP(inode);
    vnode_t *newvp;
    int error;

    VOP_OPEN(vp, &newvp, 0,
        get_current_cred(), error);

    if (error)
        return -error;

    return 0;
}
```

This is a very simple routine. XFS never returns a `newvp` and this `vnode` functionality needs more work if additional file systems are added that exploit `newvp`. For performance, XFS starts a read-ahead if the `inode` is a directory and the directory is large enough. For all cases, `xfs_open` checks to see if the file system has been shut-down and fails the open. The shutdown check provides important functionality to avoid panics and protect the integrity of the file system when errors occur such as permanent disk errors.

Example 3: The `linvfs_permission` routine

The final example is `linvfs_permission`. On linux, this routine is used to check permissions and this maps to the IRIX `VOP_ACCESS()` call as follows:

```
int linvfs_permission(struct inode *ip, int
mode)
{
    cred_t cred;
    vnode_t *vp;
    int error;

    /* convert from linux to xfs */
    /* access bits */
    mode <= 6;
    vp = LINVFS_GET_VP(ip);
    VOP_ACCESS(vp, mode, &cred, error);

    return error ? -error : 0;
}
```

2.3 `linvfs` Overhead

The overhead of the `linvfs_permission` operation has been computed. These numbers were generated by averaging 4 runs of 1000000 `access()` system calls after an initial run to warm the cache. This was performed on a 2 CPU 450 MHz PIII box.

The implementation of `VOP_ACCESS` in XFS is fundamentally the same as the permissions function code executed by `ext2`.

Table 1 shows the access system call numbers for this test. For this test, we turned off our locking code since XFS without locks more closely resembles the `ext2` code. Our goal was to measure the overhead of the `linvfs` layer, not to measure the overhead of locking.

Table 1: access() Timings

File system	Microseconds per call
ext2	3.54
xfs	3.89 (with locking commented out)

The linvfs layer overhead is around 0.35 micro-seconds for this simple call.

With locking turned on, XFS took 5.14 microseconds per call.

2.4 fcntl Versus ioctl in IRIX and Linux

In IRIX, XFS supports approximately 20 special fcntl interfaces used for space pre-allocation, extent retrieval, extended file information, etc. In addition, IRIX XFS has about a dozen special system call interfaces, all implemented via the special `syssgi` system call. These interfaces are used for operations such as growing the file system or retrieving internal file system information.

The Linux file system interface has no fcntl operation. The only supported fcntl calls on Linux are file locking calls. We proposed to the Linux community that a fcntl file operation be added. After extensive discussion, it was decided to use the existing ioctl operation, `linvfs_ioctl`, and we are in the process of converting all of the fcntl and `syssgi` usages into ioctls. A shortcoming to the ioctl approach is in the semantics of an ioctl to block or character special devices which reside within the file system: In these cases, the device driver's ioctl routine will be used rather than the file system's. Outside of that, porting the fcntl and `syssgi` interfaces to ioctl's has been straightforward.

2.5 IRIX XFS creds and Linux

In older UNIX systems, the file system code used the current process's data structures to determine the user's credentials such as uid, gid, capabilities, etc. The VFS/vnode interface removed this code and introduced a cred structure which is passed to certain file system operations such as create and lookup. The file system uses this information to determine permissions and ownership.

XFS was written using the VOP/vnode interface, so it regularly uses cred structures. One of the more prevalent cred usages on IRIX XFS is `get_current_cred`, which returns this structure for the current process.

Linux is similar to older UNIX implementations in that file systems are expected to look directly at the task

structure to determine the current process's credentials. Linux does not utilize a cred structure.

In porting XFS to Linux, we first attempted to map the various cred fields onto the corresponding task fields. This had the undesired side-effect of producing code that utilized a cred pointer that in actuality was pointing at a task. This was determined to be unacceptable.

We then considered implementing a complete cred infrastructure, which would include a pool of active creds, cred setup, teardown, lookup, etc. It was determined that this would require too much overhead.

In looking at the Linux code, we saw that all of the access/permission work occurs above the file system dependent code, so having a cred is important only on creation. We then examined our own internal usage of cred fields in XFS, and found that more often than not, a cred was passed down through a VOP_, and never used. The few places that did use a cred field were changed to use the current task structure in Linux.

We still pass a cred address on the VOP_s, but it is not used. In the future, as part of the port cleanup, we may change the VOP_ macros, or more likely simply pass NULL as the cred address.

In addition to these cred changes, we have removed many access checks from the XFS code since these are now performed at a higher layer and are redundant in the file system dependent code.

3. XFS Caching and I/O

When XFS was first implemented within IRIX, the buffer cache was enhanced in a number of ways to better support XFS, both for better file I/O performance and for better journaling performance. The IRIX implementation of XFS depends on this buffer cache functionality for several key facilities.

First, the buffer cache allows XFS to store file data which has been written by an application without first allocating space on disk. The routines which flush delayed writes are prepared to call back into XFS, when necessary, to get XFS to assign disk addresses for such blocks when it is time to flush the blocks to disk. Since delayed allocation means that XFS can determine if a large number of blocks have been written before it allocates space, XFS is able to allocate large extents for large files, without having to reallocate or fragment storage when writing small files. This facility allows XFS to optimize transfer sizes for writes, so that writes can pro-

ceed at close to the maximum speed of the disk, even if the application does its write operations in small blocks. In addition, if a file is removed and its written data is still in delayed allocation extents, the data can be discarded without ever allocating disk space.

Second, the buffer cache provides a reservation scheme, so that blocks with delayed allocation will not result in deadlock. If too much of the available memory is used for delayed allocation, a deadlock on the memory occurs when trying to do conversion from delayed to real allocations. The deadlock can occur since the conversion requires metadata reads and writes which need available memory.

Third, the buffer cache and the interface to disk drivers support the use of a single buffer object to refer to as much as an entire disk extent, even if the extent is very large and the buffered pages in memory are not contiguous. This is important for high performance, since allocating, initializing, and processing a control block for each disk block in, for example, a 7 MB HDTV video frame, would represent a large amount of processor overhead, particularly when one considers the cost of cache misses on modern processors. XFS has been able to deliver 7 GB/second from a single file on an SGI Origin 2000 system, so the overhead of processing millions of control blocks per second is of practical significance.

Fourth, the buffer cache supports “pinning” buffered storage in memory, which means that the affected buffers will not be written to disk until they have been “unpinned”. XFS uses a write-ahead log protocol for metadata writes, which means XFS writes a log entry containing the desired after-image before updating the actual on disk metadata. On recovery, XFS just applies after-images from the log (in case some of the metadata writes were not completed). In order to avoid having to force the log before updating metadata, XFS “pins” modified metadata pages in memory. Such pages must count against the memory reservation (just as do delayed allocation pages). XFS pins a metadata page before updating it, logs the updates, and then unpins the page when the relevant log entries have been written to disk. Since the log is usually written lazily, this in effect provides group commit of metadata updates.

With Linux 2.3 and later releases, the intent is that most file system data will be buffered in the page cache, but the I/O requests are still issued one block at a time, with a separate `buffer_head` for each disk block and multiple `buffer_head` objects for each page (if the disk block size is smaller than the page size). As in Linux 2.2, drivers may freely aggregate requests for adjacent disk blocks

to reduce controller overhead, but they must discover any possibilities for aggregation by scanning the `buffer_head` structures on the disk queue.

3.1 The pagebuf Module

Our approach to porting XFS has included adding `pagebuf`, a layered buffer cache module on top of the Linux page cache. This allows XFS to act on extent-sized aggregates. Key to this approach is the `pagebuf` structure, which is the major structure of the `pagebuf` layer. The `pagebuf` objects implemented by this module include a *kiobuf* (a Linux data structure which describes one or more lists of physical pages) to describe the set of pages (or parts of pages) associated with `pagebuf`, plus the file and device information needed to perform I/O. We are experimenting with a new device request interface, so that we can queue one of these `pagebuf` objects directly to a device, rather than having to create and queue a large number of single-block `buffer_head` objects for each logical I/O request. For backward compatibility, we support a routine which does create and queue `buffer_head` objects to perform the I/O for `pagebuf`.

A key goal for the layered buffer cache module is that its objects be strictly temporary, so that they are discarded when released by the file system, with all persistent data held purely in the page cache. This requires storing a little more information in each `mem_map_t` (page frame descriptor), but it avoids creating yet another class of permanent system object, with separate locking and resource management issues. The IRIX buffer cache implementation has about 11000 lines of very complex code. By relying purely on the page cache for buffering, we avoid most of the complexity, particularly in regard to locking and resource management, of hybrid page and buffer caches, at the cost of having to pay careful attention to efficient algorithms for assembling large buffers from pages.

3.2 Partial Aggregate Buffers

Pages within an extent may be missing from memory, so a buffer for an extent may not find all pages present. If the buffer is needed for writing, empty (invalid) pages may be used to fill the holes, in cases where the intended write will overwrite the missing pages. If only part of a page will be modified, `pagebuf` will read in the missing page or pages. If the buffer is needed for reading or writing just part of the extent, missing pages need not be read if all pages to be read are present.

XFS extents are typically large. However, small extents are still possible, and hence a page could be mapped by

more than one extent. In such cases, it would be desirable to avoid having to seek to multiple locations when the I/O is to a part of the page mapped by one extent. Therefore, we added a map (`block_map`) of valid disk blocks for each page. When the entire page is read or written, all blocks of the page are marked valid. On a file write, if a part of the page is modified, only the corresponding bits for the modified blocks are marked valid. Similar actions are performed during a read of a partial page. Note that similar information is maintained by Linux if buffers are mapped (or not mapped) to parts of the page. `block_map` provides this functionality without having to attach `buffer_head`'s to the page. Finally, if the page and the disk block sizes are the same, the `block_map` is equivalent to (and is replaced by) the `PG_uptodate` flag of the page.

3.3 Locking for pagebufs

As noted above, the basic model for pagebufs is that each pagebuf structure is independent, and that it is just a way of describing a set of pages. In this view, there may be multiple pagebufs referring to a given page, just as a given page may be mapped into multiple address spaces. This works well for file data, and allows pagebufs in the kernel to be operationally equivalent to memory mapped file pages in user mode. For metadata, however, XFS assumes that its buffer abstraction allows for sleeping locks on metadata buffers, and that metadata buffers are unique. That is, no two metadata buffers will share a given byte of memory. On the other hand, two metadata buffers may well occupy disjoint portions of a single page.

To support this model, we implemented a module layered above the basic pagebuf module, which keeps an ordered list (currently an AVL tree) of active metadata buffers for a given mounted file system. This module ensures uniqueness for such buffers, assures that they do not overlap, and allows locking of such buffers. Since we were unsure whether this facility would be of use for other file systems, we have kept it separate from the basic pagebuf module, to keep the basic module as clean and fast as possible.

3.4 Delayed Allocation of Disk Space for Cached Writes

Allocating space when appending to a file slows down writes, since reliable metadata updates (to record extent allocations) result in extra writes. Also, incremental allocations can produce too-small extents, if new extents are allocated each time a small amount of data is appended to a file (as when many processes append to a

log file). Delayed allocation reserves disk space for a write but does not allocate any particular space; it simply buffers the write in the page cache. Later, when pages are flushed to disk, the page writing path must ask the file system to do the actual allocation. Also, to allow for optimal extent allocations and optimal write performance, the page writing path must collect adjacent dirty pages ("cluster" them) and write them as a unit.

Since allocation of disk space may be required in the page writing path when delayed allocation is present, and such allocation may require the use of temporary storage for metadata I/O operations, some provision must be made to avoid memory deadlocks. The delayed allocation path for writes must make use of a main memory reservation system, which will limit the aggregate amount of memory used for dirty pages for which disk space has not been allocated, so that there will always be some minimum amount of space free to allow allocations to proceed. Any other non-preemptible memory allocations, such as kernel working storage pages, must be counted against the reservation limit, so that the remaining memory is genuinely available. Based on experience with IRIX, limiting dirty delayed allocation pages and other non-preemptible uses to 80% of available main memory is sufficient to allow allocations to proceed reliably and efficiently.

Page flushing should flush enough delayed allocation pages to keep some memory available for reservation, even if there is free memory. That is, the page flushing daemon must attempt both to have free memory available and have free reservable memory available. The reservation system must, of course, allow threads to wait for space.

3.5 Page Cleaning

At present in Linux, a dirty page is represented either by being mapped into an address space with the "dirty" bit set in the PTE, or by having `buffer_head` objects pointing to it queued on the buffer cache delayed write queue, or both. Such dirty pages are then eventually written to disk by the `bdflush` daemon. This has several drawbacks for a high-performance file system. First, having multiple `buffer_head` objects to represent, in effect, one bit of state for page is inefficient in both space and time. Second, when dirty pages are cleaned, there is no particular reason to expect that all adjacent dirty pages within an extent will be written at once, so the disk is used less efficiently. Third, modifications to pages made via `mmap()` may be indefinitely delayed in being written to disk, so many updates may be lost on a power failure, unless `msync()` is used frequently. That is, either one

slowly and synchronously updates the disk, or one gives up any expectation of timely updating of the disk. Fourth, disk write traffic is not managed for smoothness, so there is considerable burstiness in the rate of disk write requests. This in turn reduces utilization of the drive and increases the variability of read latencies.

For the XFS port, we are implementing a page cleaner based on a “clock” algorithm. If the system is configured to move a given update to disk within K seconds, and if there are N dirty pages out of M total pages, the page cleaner will visit enough pages every second to clean N/K pages. That is, the page cleaner will advance its clock hand through the page frame table (`mem_map`) until it has caused sufficient pages to be queued to the disk to meet its target. Since the page write path will cluster dirty pages, visiting a given dirty page may cause multiple pages to be written, so the page cleaner may not directly visit as many dirty pages as are written.

Note that the page cleaner does not simply run once a second. Rather, it runs often enough (multiple times per second) to maintain a queue of disk writes, but without creating an excessively long queue (which would increase the variability of read latency). Moreover, on a ccNUMA system, there will be a page cleaner thread per node, so that page cleaning performance will scale with the size of the system.

Since the page cleaner must steadily clean pages, it must not block for I/O. For clusters of pages where disk space has already been allocated, the page cleaner can simply queue a pagebuf to write the cluster to disk. For cases where the allocation has been delayed, however, the allocation of disk space by the file system will in general require waiting to read in metadata from disk and may also require waiting for a metadata log write, if the log happens to be full. The page cleaner has a number of writeback daemon threads to handle writing pages for which disk allocation has been deferred. (This could also be used by any file system where a write might block.) The file system `pagebuf_iostart` routine has a `flags` argument, and a flag, `PBF_DONTBLOCK`, which the page cleaner uses to indicate that the request should not block. If the `pagebuf_iostart` routine returns an `EWouldBlock` error, the page cleaner queues the cluster pagebuf to its writeback daemon threads, one of which then processes it by calling the file system `pagebuf_iostart` routine without the `PBF_DONTBLOCK` flag.

3.6 Pinning memory for a buffer

In the course of the XFS port, it became clear that metadata objects would in many cases be smaller than pages. (This will be even more common on systems with larger page sizes.) This means that we cannot actually “pin” the entire page, since doing so might keep the page from being cleaned indefinitely: The different metadata objects sharing the page might be logged at various times, and at least one might be pinned at any given time. We decided to treat metadata pages specially in regard to page cleaning. The page cleaning routine associated with a metadata page (via the `address_space` operations vector) looks up any metadata pagebufs for the page, instead of simply writing out the page. Then, for each such metadata pagebuf which is not “pinned”, the page cleaning routine initiates writing the pagebuf to disk. The page cleaning routine simply skips any pinned pagebufs (which will then be revisiting on the next page cleaning cycle).

Since only metadata pages are pinned, we do not need a separate mechanism to pin regular file data pages. If, however, it were desirable to integrate transactional updates for file data into a future file system, it would be possible to simply add a “pin” count to the `mem_map_t`. We have not done this, however, since XFS does not support transactional update of file data.

3.7 Efficient Assembly of Buffers

We would like the overhead for finding all valid pages within an extent to be low. At present, we simply probe the hash table to find the relevant pages. If this proves to be a performance problem for large I/O requests, we could modify the `address_space` object to record pages in some sorted fashion, such as an AVL tree, so that we could quickly locate all of the pages in a range, at a very low and constant cost after the first one.

3.8 File I/O

The Linux XFS file read and write routines are provided by the default Linux I/O path with calls into `page_buf`, when appropriate.

Buffered File Reading

The read routine first loops through the user’s I/O requests, searching for pages by probing the page cache. If a page is present, the read routine copies the data from the page cache to the user buffer. If all of the pages are present, the read is then complete.

When a page is not found, what the read routine will do depends on the remaining I/O size. If the remaining I/O

size is less than the page cache size times 4 (which is 16K for IA 32 systems, which have a 4K page size) the default Linux kernel `do_generic_file_read` is performed, just as for an ext2 file system.

If the remaining I/O size when a page is not found is greater than 4 times the page cache size, the read routine calls back into the file system via a `pagebuf_bmap` routine to obtain an extent map covering the portion of the file to be read which is not in the cache. The extent map is composed of one or more extent descriptions. Each can be a real extent (with a disk address and a length), a hole (with no disk storage assigned or reserved), or a delayed allocation (with disk storage reserved but not assigned).

Unlike the conventional Linux bmap file system call-back routine, the `pagebuf_bmap` routine can return a list of extentmaps, not simply a single disk block address. Each entry in the list of extentmaps can represent many file system blocks.

One or more instances of an extent map are returned by `pagebuf_bmap()`. Table 2 shows the fields that are contained in an extent map.

Table 2: Contents of Extent Map

Field	Meaning
bn	starting block number of map
offset	byte offset from bn (block number) of user's request.
bsize	size of mapping in bytes
flags	option flags for mapping

Table 3 shows the option flags that are currently used in an extent map.

Table 3: Option Flags for Mapping

Flag	Meaning
HOLE	mapping covers a hole
DELAY	mapping covers dealloc region
NEW	mapping was just allocated
no value	mapping is actual disk extent

For non-delayed allocations, the pages might or might not be present in the cache. For a hole, any cached pages

would have been created by `mmap` access or by small reads. The `pagebuf` read path does not allocate pages for holes. Rather, it zeros out the user's buffer.

If a read returns a delayed allocation, the pages must exist and are copied to the user's buffer.

If the map is not HOLE or Delay, the read routine fills in the disk block addresses, allocates empty pages and enters them in the page cache, attaches them to the `pagebufs`, and queues the read requests to the disk driver. A page may already be marked as being up to date, however, in which case it is not necessary to read that page from disk.

After the requests have completed, the read routine marks the pages valid, releases the `pagebufs`, and finishes copying the data to the user buffer. If a single `pagebuf` is not sufficient, the read path waits for the data from the first `pagebuf` read requests to be copied to the user buffer before getting the next `pagebuf`.

Buffered File Writing

The write routine, like the read routine, first loops through the file page offsets covering the desired portion of the file, probing the page cache. If a page is present, the write routine copies the data from the user buffer to the page cache, and marks the pages dirty. If `O_SYNC` is not set, the write is then complete. If `O_SYNC` is set, the write routine writes the pages synchronously to disk before returning.

If a page is not found, the write routine calls back into the file system to obtain the extent map that covers the portion of the file to be written. The call to the file system is for the offset and length of the user's I/O, rounded to page boundaries.

The file system `pagebuf_bmap` routine accepts some optional flags in a flag argument. One, `PBF_WRITE`, specifies that a write is intended. In this case, any hole will be converted to a real or delayed allocation extent. A second flag, `PBF_ALLOCATE`, specifies that a real extent is required. In this case, any delayed allocation extent will be converted to a real extent. This latter flag is used when `O_SYNC` is set, and is also used when cleaning pages or when doing direct I/O.

The `NEW` flag indicates when a mapping has just been allocated, and should not be read from disk. This can occur when a hole is converted to allocated space, or you are allocating new space at the end of a file. Empty pages are then allocated and hashed for any pages not

present in the cache. Any pages covering a new map which will be partially overwritten are then initialized to zero, at least insofar as they will not be overwritten. If the NEW flag is not set, the space already existed in the file. In this case, any pages which will only be partially overwritten must be synchronously read from disk before modification with the new user's data.

In both the NEW and already existing cases, the pages are marked as up to date but dirty, indicating that they need to be written to disk.

Preliminary I/O Performance Testing

We have tested the read performance for XFS on Linux, using the `lmdd` program of the `lmbench` toolkit to time the I/O. We used `lmdd` I/O request sizes from 1K to 1024K for the following file system configurations. Each file system was created with a 4K block size.

- XFS file system that does not use pagebufs
- XFS file system with 16K cutover point for using pagebufs
- XFS file system with 32K cutover point for using pagebufs
- ext2 file system

What we found for this specific test and configuration was that for XFS file systems, currently the I/O size makes only a small difference in performance. We also have not yet seen a significant difference in performance when switching the pagebuf cutoff size between 16K (4 pages) and 32K (8 pages). This was because the test was disk bound. At the time of this writing, we have done only minimal performance testing on a single thread without read-ahead. We expect to see better performance with multiple threads and faster disks.

For XFS and the specific configuration tested, the file system I/O rate generally fluctuated between approximately 19.01 and 19.09 MB/sec. For an ext2 file system using the same test, the file system I/O rate fluctuated between approximately 18.8 and 18.9 MB/sec.

3.9 Direct I/O

Small files which are frequently referenced are best kept in cache. Huge files, such as image and streaming media files and scientific data files, are best not cached, since blocks will always be replaced before being reused. Direct I/O is raw I/O for files: I/O directly to or from user buffers, with no data copying. The page cache must cooperate with direct I/O, so that any pages, which are cached and are modified, are read from memory, and so that writes update cached pages.

Direct I/O and raw I/O avoid copying, by addressing user pages directly. The application promises not to change the buffer during a write. The physical pages are locked in place for the duration of the I/O, via the Linux `kiobuf` routines. The read and write paths for the pagebuf module treat direct I/O requests much as they do regular requests, except for identifying the pages to be addressed in the pagebuf. That is, for direct I/O, the user buffer is referenced via a `kiovec`, which is then associated with the pagebuf, instead of locating page cache pages and addressing them via a `kiovec`.

Any dirty pages in the page cache must be flushed to disk before issuing direct I/O. The normal case will find no pages in the cache, and this can be efficiently tested by checking the inode. Once the pagebuf is assembled, the I/O path is largely common with the normal file I/O path, except that the write is never delayed and allocation is never delayed.

Direct I/O is indicated at `open()` time by using the `O_DIRECT` flag. Usually the needed space for the file is pre-allocated using an XFS `ioctl` call to insure maximum performance.

4. Volume Management Layers

The integration of existing Linux volume managers with the XFS file system has created some issues for the XFS port to Linux.

Traditional Linux file systems have been written to account for the requirements of the block device interface, `ll_rw_block()`. `ll_rw_block` accepts a list of fixed size I/O requests. For any given block device on a system, the basic unit of I/O operation is set when the device is opened. This size is then a fixed length of I/O for that device. The current implementations of Linux volume managers have keyed off this fixed size I/O and utilize an I/O dispatcher algorithm.

By using a fixed I/O length, the amount of "math" that is needed is significantly less than what it would be if the I/O length were not fixed. All I/O requests from a file system will be of the same size, as both metadata and user data is of fixed size. Therefore, all underlying devices of a logical volume must accept I/O requests of the same size. All that the volume manager needs to do for any I/O request is to determine which device in the logical volume the I/O should go to and recalculate the start block of the new device. Each I/O request is directed wholly to a new device.

The XFS file system, however, does not assume fixed size I/O. In an XFS file system, metadata can be anywhere from 512 bytes to over 8 Kbytes. The basic minimum I/O size for user data is set at file system creation time, with a typical installation using 4 Kbytes. One of the XFS design goals was to aggregate I/O together, creating large sequential I/O.

This feature of XFS creates a problem for current Linux volume managers, since the XFS file system can hand an I/O request off to a block device driver specifying the start position and length, which is not always fixed. A logical volume manager is just another block device to XFS, and a logical volume manager working in conjunction with XFS needs to be able to handle whatever size I/O request XFS desires, to some reasonable limit.

One of the options to address this problem in XFS is to change the on disk format of the file system to use a fixed size. This would render the Linux version of XFS incompatible with the current IRIX implementations, however, and so it was deemed unacceptable, just as making different versions of NFS would be unacceptable.

Currently, XFS for Linux is addressing the problem of variable I/O request size by opening a device with the minimum I/O size needed: 512 bytes. Any request calling `ll_rw_block` directly must be of that basic size. User data requests use a different version of `ll_rw_block`, that has been modified to accept a larger size.

Since all Linux volume managers use a call out from `ll_rw_block` it is clear that XFS currently will not work with current implementations.

In the long run, the solution to the problem of using XFS with currently available Linux volume managers is not clear. Changing interfaces in the kernel that other file systems rely on is not an easy thing to do. It requires agreement of all the current file-system maintainers to change their interface to the kernel.

One thing that is clear is that it will be necessary to develop an additional layer above `ll_rw_block` that accepts I/O requests of variable size. This interface would either be a direct connection to a device driver, an interface to a logical volume manager, or the fall back case of just calling `ll_rw_block` for compatibility.

The logical volume interface may need to "split" the I/O request into multiple sub-I/O requests if the I/O is large enough to span multiple devices. This not a difficult

thing to implement, but it must be done in agreement of all the logical volume maintainers.

Since it is becoming apparent that Linux is growing up and moving to larger and higher performance hardware. The high bandwidth I/O that XFS offers will be needed. High performance logical volumes will be an integral part of this.

5. Moving XFS to Open Source

For XFS to be a viable alternative file system for the open source community, it was deemed essential that XFS be released with a license at least compatible with the GNU Public License (GPL).

The IRIX operating system in which XFS was originally developed has evolved over a long period of time, and includes assorted code bases with a variety of associated third party license agreements. For the most part these agreements are in conflict with the terms and conditions of the GNU Public License.

The initial XFS project was an SGI initiative that started with a top-to-bottom file system design rather than an extension of an existing file system. Based upon the assertions of the original developers and the unique features of XFS, there was a priori a low probability of overlap between the XFS code and the portions of IRIX to which third-party licenses might apply. However it was still necessary to establish that the XFS source code to be open sourced was free of all encumbrances, including any associated with terms and conditions of third party licenses applying to parts of IRIX.

SGI's objectives were:

- to ensure the absence of any intellectual property infringements
- to establish the likely derivation history to ensure the absence of any code subject to third party terms and conditions

This was a major undertaking; as the initial release of buildable XFS open source contained some 400 files and 199,000 lines of source. The process was long, but relatively straightforward, and encumbrance relief was usually by removal of code.

The encumbrance review was a combined effort for SGI's Legal and Engineering organizations. The comments here will be confined to the technical issues and techniques used by the engineers.

5.1 The Encumbrance Review Process

We were faced with making comparisons across several large code bases, and in particular UNIX System V Release 4.2-MP, BSD4.3 NET/2, BSD4.4-lite and the open source version of XFS. We performed the following tasks:

1. Historical survey

We contacted as many as possible of the original XFS developers and subsequent significant maintainers, and asked a series of questions. This information was most useful as guideposts or to corroborate conclusions from the other parts of the review.

2. Keyword search (all case insensitive)

In each of the non-XFS code bases, we searched for keywords associated with unique XFS concepts or technologies (e.g. journal, transaction, etc.). In the XFS code base, we searched for keywords associated with ownership, concepts and technologies in the non-XFS code bases (e.g. at&t, berkeley, etc.).

3. Literal copy check

Using a specially built tool, we compared every line of each XFS source file against all of the source in the non-XFS code bases. The comparison ignored white space, and filtered out some commonly occurring strings (e.g. matching “i++;” is never going to be helpful).

4. Symbol matching

We developed tools to post-process the ASCII format databases from cscope to generate lists of symbols and their associated generic type (function, global identifier, macro, struct, union, enum, struct/union/enum member, typedef, etc.). In each XFS source file the symbols were extracted and compared against all symbols found in all the non-XFS code bases. A match occurred when the same symbol name and type was found in two different source files. Some post-processing of the symbols was done to include plausible name transformations, e.g. adding an “xfs_” prefix, or removal of all underscores, etc.

5. Prototype matching

Starting with a variant of the mkproto tool, we scanned the source code to extract ANSI C prototypes. Based on some equivalence classes, “similar” types were mapped to a smaller number of base

types, and then the prototypes compared. A match occurred when the type of the function and the number and type of the arguments agreed.

6. Check for similarity of function, design, concept or implementation.

This process is based upon an understanding, and a study, of the source code. In the XFS code, for each source file, or feature implemented in a source file, or group of source files implementing a feature, it was necessary to conduct a review of the implementation of any similar source file or feature in each of the non-XFS code bases. The objective of this review is to determine if an issue of potential encumbrance arises as a consequence of similarity in the function, implementation with respect to algorithms, source code structure, etc.

7. Check for evidence of license agreements.

We examined the XFS code (especially in comments) to identify any references to relevant copyrights or license agreements.

In all of the steps above, the outcome was a list of *possible* matches. For each match, it was necessary to establish in the context of the matches (in one or more files), if there was a real encumbrance issue.

We used a modified version of the tkdifff tool to graphically highlight the areas of the “match” without the visual confusion of all of the minutiae of the line-by-line differences. However, the classification of the matches was ultimately a manual process, based on the professional and technical skills of the engineers.

5.2 Encumbrance Relief

Especially in view of the size of the XFS source, a very small number of real encumbrance issues were identified.

In all cases the relief was relatively straightforward, with removal of code required for IRIX, but not for Linux, being the most common technique.

6. Summary

Porting the XFS file system to Linux required that we address a variety of technical and legal issues. In this paper, we have summarized how the file and inode operations of the XFS vnode interface were translated to Linux by means of the linvfs layer. We have also described how XFS caching was implemented in Linux, and how this caching is used for I/O operations. Finally,

we provided a brief overview of the volume management layer of XFS, and how this has been implemented in Linux.

In addition to these technical concerns we have summarized the technical aspects of the encumbrance review process we went through to ensure that XFS could be released to the open source community without legal ramifications.

We continue to evaluate our use of the `linvfs` porting layer, especially as regards performance. Our new buffer layer has addressed many of the performance issues and generally extended Linux functionality. XFS on Linux is evolving and will meet the demands of applications moving from traditional UNIX platforms to Linux.

7. Availability

We have completed our initial port of XFS to Linux and the open source process for that port and we are currently in the process of finishing the work on the code. The current version of XFS for Linux is available for downloading at <http://oss.sgi.com/projects/xfs>.

References

- [1] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto and Geoff Peck. *Scalability in the XFS File System*. <http://www.usenix.org/publications/library/proceedings/sd96/sweeney.html>, January 1996.
- [2] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pp. 238–247, June 1986.
- [3] Jim Mostek, William Earl, Dan Koren, Russell Cattelán, Kenneth Preslan, and Matthew O’Keefe. *Porting the SGI XFS File System to Linux*. <http://oss.sgi.com/projects/xfs/papers/als/als.ps>, October 1999.

LinLogFS — A Log-Structured Filesystem For Linux

Christian Czeatzke

xS+S

{czeatzke,anton}@mips.complang.tuwien.ac.at

<http://www.complang.tuwien.ac.at/projects/linux.html>

M. Anton Ertl

TU Wien

Abstract

LinLogFS is a log-structured filesystem for Linux. It currently offers features like fast crash recovery and in-order write semantics. We implemented LinLogFS by putting a logging layer between an adapted version of the ext2 file system and the block device.

1 Introduction

In early 1998 we started working on LinLogFS¹ to address the two main shortcomings that Linux had (in comparison to Digital/Tru64 Unix with AdvFS):

- Crash recovery is slow and sometimes requires manual intervention.
- There is no way to make *consistent* backups while the file system is writable.

These shortcomings are not just inconvenient for the users of desktop machines, they are also one of the main barriers against Linux servers in applications requiring high availability.

We decided to solve these problems by implementing a log-structured file system; this approach also promises a number of other benefits:

- In-order semantics for data consistency upon system failure (see Section 3.2).
- Cheap cloning (writable snapshots) of the whole file system can be used for exploring what-if scenarios while writing to the other clone of the file system (in addition to consistent backups).

¹Formerly known as dtfs (renamed to avoid conflicts with SCO's dtfs).

- Growing, shrinking, or migrating a file system while it is mounted.
- Fast update of a network mirror after a network failure.
- Relatively good performance for synchronous writes (NFS).
- Good write performance with RAID5s.

In the meantime, there are a number of Linux projects nearing completion that address fast crash recovery and consistent backups (see Section 8), and may also provide some of the other advantages. However, we think that LinLogFS provides an interesting design point, and that its features combined with its performance will make it attractive once it is mature.

We first introduce log-structured file systems (Section 2), then discuss some advanced topics in the design of log-structured file systems (Section 3). In Section 4 we describe which of these ideas we have implemented until now. Then we describe our implementation, first the layering (Section 5), then the modifications to the Linux kernel (Section 6). In Section 7 we compare the performance of LinLogFS and other Linux file systems. In Section 8 we discuss related work.

2 A Log-Structured File System

This section explains the concept of log-structured file systems by using LinLogFS as an example. Most of this description also fits other log-structured file systems like BSD LFS. Section 8 describes the differences from other systems and the original points in LinLogFS.

A file system provides a layer of abstraction that allows the user to deal with files and directories on top of a block-oriented storage medium. In Unix the

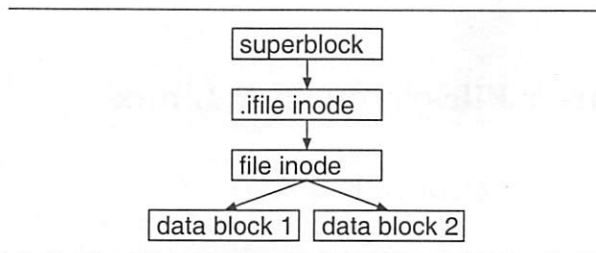


Figure 1: On-disk structure for locating data blocks of a file

file system can be considered as consisting of two layers: the upper layer maps the hierarchical structure of the directory tree onto the flat namespace of inode numbers; the lower layer maps an inode number and a position within a file to the disk block that contains the data. The difference between log-structured and other file systems is in the lower layer, so we will focus on it in this section (and in the rest of this paper).

2.1 Reading

Reading is quite similar to conventional file systems like ext2 and BSD FFS: when reading a certain block from a certain file, the file's inode is accessed, and the data block is accessed by following the pointer there (possibly going through several levels of indirect blocks).

How do we find the inode? In LinLogFS (in contrast to conventional file systems) the inodes reside in the file `.ifile`. We find the inode of this file through the superblock (see Fig. 1).

2.2 Writing

The distinctive feature of log-structured file systems is that they perform no update-in-place; instead they write changed data and meta-data to an unused place on the disk. So, when writing a block of data to a file, the block is written to a new location, the now-changed indirect block pointing to it is written to a new location, and so on for the whole chain from the superblock to the data block (see Fig. 2).²

Benefit: Since existing data is not destroyed by writing over it, it is easy to clone the file system for backup purposes and to undelete files.

²This is similar to the handling of data structures in single-assignment languages, in particular eager functional programming languages like ML.

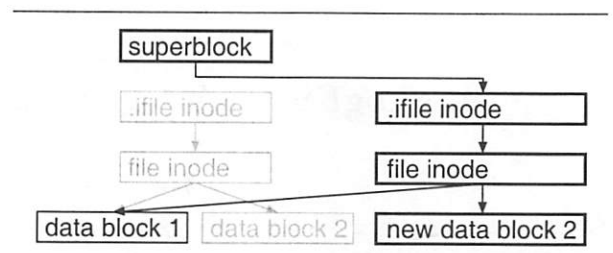


Figure 2: Change of the on-disk structure when overwriting data block 2 of the file

The superblock is written and updated in a fixed place, allowing us to find it, the `.ifile` and thus the whole file system when mounting it. Therefore, updating the superblock commits all the writes since the last superblock update.

Benefit: Having explicit commits allows performing atomic operations involving more than one block, e.g., directory operations. This makes it easy to ensure that the committed state of the file system is consistent and provides fast crash recovery. It also makes it easy to guarantee in-order write semantics upon a crash (see Section 3.2).

Since there is no restriction on where the blocks must be written, they can all be appended to a log. Most of the time a number of writes can be collected and written in one large batch. Ideally the disk is written from the start to the end (see Section 2.3 for deviations from this ideal).

Benefit: The resulting sequential writes require no disk seeks (except for the superblock updates; see Section 3.1.1 for reducing these seeks) and thus allow fast synchronous writes. Large sequential writes are also helpful when writing to RAIDs. Moreover, they make it easy to find the newly written data when synchronizing a (network) mirror.

2.3 Reclaiming Space

Unfortunately disks are finite, so we have to reclaim the space occupied by deleted or overwritten data before the disk is full. LinLogFS divides the volume into segments of about 512KB.³ The *cleaner* program reclaims a segment by copying the (hopefully small amount of) live data to the end of the log.

³Note that starting a new segment does not commit changes written in the last one (at least in LinLogFS).

Then the reclaimed segment is available again for writing.

Benefit: Using large segments instead of a free blocks map ensures that the filesystem can usually write sequentially, avoiding seeks. Using segments instead of treating the log as one big ring buffer allows segregating long-lived from short-lived data and varying the cleaning frequency accordingly. The copying approach of the cleaner makes it easy to free specific segments in order to shrink or migrate a volume.

The cleaner uses segment summary information: For every block in the segment, the segment summary contains the file and the block number within the file. This makes it efficient for the cleaner to find out whether a block is live.

3 Log-structured File System Details

This section discusses a number of design issues for log-structured file systems in more detail.

3.1 Optimizations

3.1.1 Roll forward

Writing the superblock requires a seek to the superblock and a seek back to the log, limiting the performance for small synchronous writes.

This can be solved by writing special commit blocks into the log. When mounting a file system, the file systems does not assume that the superblock points to the end of the log. Instead, it uses the pointed-to block as a roll-forward start point and scans forward for the last written commit block. The file system uses that commit block as the end of the log; i.e., it contains the `.ifile` inode and allows finding the whole file system. To limit the crash recovery time, the superblock is still written now and then (e.g., when about 10MB of segments have been written to achieve a crash recovery time on the order of one second).

3.1.2 Change Records

Instead of writing every changed block out when committing, space and write time can be saved in many cases by recording just what has changed.

During roll-forward, the file system updates its in-memory copies of the changed (meta-)data with this information. Of course, the changed blocks still have to be written out when establishing a new roll-forward start point, but for blocks that are changed several times between such start points, this optimization pays off.

The state to which a change record is applied on recovery is known completely, so we can use any kind of change record (in contrast to journaling file systems, where change records are usually restricted to idempotent operations [VGT95]).

Change records influence recovery time by requiring reading the blocks that they change (and possibly indirect blocks to find them). This should be taken into account when deciding when to write a new roll-forward start point. Another consequence is that the cleaner has to treat these blocks as live.

3.1.3 Block Pointers

Writing one data block can cause up to eight meta-data blocks to be written to the log: up to three indirect blocks, the block containing the inode of the file, up to three indirect blocks of the `.ifile`, and the commit block containing the `.ifile` inode; the reason for this cascade is that the pointer to the next block in the chain changes. The usual case is much better, because usually many written data blocks share meta-data blocks. However, for small synchronous writes block pointer updates would be a problem. Moreover, part of this meta-data (the `.ifile` inode and its indirect blocks) tends to be overwritten and thus become dead quickly, requiring a cleaning pass even if none of the user data in the segment has died.

To alleviate this problem, we are considering to add change records for the most frequent cause of meta-data updates: updates of pointers to blocks. The change record would describe which block of which file now resides where.

There is an improvement of this optimization: we need to store the information about which blocks reside where only once, not once for the change records and once as segment summary information for the cleaner.

Another refinement of this optimization is to combine the change records for ranges of blocks of the same file into one change record, saving space.

3.2 In-order semantics

With *in-order semantics* we mean that the state of the file system after recovery represents all write(s) (or other changes) that occurred before a specific point in time, and no write() (or other change) that occurred afterwards. I.e., at most you lose a minute or so of work.

The value of this guarantee may not be immediately obvious. It means that if an application ensures file data consistency in case of its own unexpected termination by performing writes and other file changes in the right order, its file data will also be consistent (but possibly not up-to-date) in case of a system failure; this is sufficient for many applications.

However, most file systems nowadays guarantee only meta-data consistency and require the extensive use of `fsync(2)` to ensure any data consistency at all. So, if applications `fsync`, isn't the in-order guarantee worthless? Our experience (<http://www.complang.tuwien.ac.at/anton/sync-metadata-updates.html>) suggests that even popular applications like Emacs don't `fsync` enough to avoid significant data losses; and even for applications that try to ensure data consistency across OS crashes with `fsync`, this is probably not a very well-tested feature. So, providing the in-order guarantee will improve the behaviour of many applications.

Moreover, `fsync()` is an expensive feature that should not be used when cheaper features like the in-order guarantee are sufficient.

One problem with guaranteeing in-order semantics in a log-structured file system is that frequent `fsyncs` can lead to fragmenting unrelated files.⁴ There are two possible solutions:

- Defragment the files upon cleaning.
- Weaken the guarantee such that the state of the file system after recovery represents all write(s) (or other changes) that occurred before a specific point in time, and only `fsync()`s that occurred afterwards; after all, applications using `fsync` supposedly know what they are doing.

⁴The reason for this fragmentation is: with in-order semantics, `fsync()` is equivalent to `sync()`, i.e., all changes are written out. Now assume a workload where lots of files are written concurrently. Given the sequential writing strategy of log-structured file systems, parts of unrelated files end up close to each other, and these files will be fragmented.

3.3 Cleaning and Cloning

We have not implemented a cleaner for LinLogFS yet; this section reflects our current thoughts on cleaning.

The cleaning heuristics discussed in the literature [RO92, BHS95] work well, so this is the direction we would like to take: cleaning proactively when the disk is idle, and using utilization and age for choosing segments to clean. We also like the Spiralog [WBW96] idea of producing entire segments out of collected data (instead of copying the collected data into the current segment), which allows avoiding mixing up data with different ages (i.e., different expected lifetimes). The cleaner can also be used for defragmentation.

The main new problem in cleaning LinLogFS is how to deal with cloning. The cleaners described in the literature use information about the utilization of a segment in their heuristics. This information is not easily available in the presence of cloning; in particular, dismissing a clone can result in changes in utilization that are expensive to track. File deletion would also be relatively expensive (especially considering that it would be very cheap otherwise): for every block in the file the filesystem would have to check whether the block is still alive in any other clone.

Therefore, we have to search for heuristics that work without this information, or with just approximate information (like upper and lower bounds on utilization).

If we don't find such heuristics, we may have to bite the bullet, and keep track of which blocks are alive in which clone. WAFL [HLM94] uses a simple bit matrix for this (with 32 bits/block). This has the following disadvantages:

RAM consumption For a 20GB file system with 4KB blocks, this requires 20MB, much of which will have to reside in RAM at most times, in particular when cloning or dismissing clones. This is especially worrying because disks tend to grow faster than RAM since the early 1990s.

High cloning and dismissing cost Cloning requires copying a bit to another bit in each word of the matrix. This can take longer than you want to lock out file system write operations, so you may have to allow writing during cloning; also, the new matrix has to be written to disk.

Limited number of clones There can be at most

32 clones (20 in WAFL).

A more sophisticated data structure may reduce these problems; the following properties can be exploited: most blocks are alive in all clones, or dead in all clones; large extents of blocks will be all alive or all dead everywhere.

If we have per-block liveness information, it may be preferable to simply write to dead blocks instead of using a cleaner to clean segments. WAFL uses this approach, but it is unclear how well it performs without NVRAM support, especially in the presence of frequent fsyncs or syncs. It also requires some sophistication in selecting where to write.

3.4 Write Organization

We have not described how LinLogFS organizes the blocks within a segment, because we intend to change this (the current organization is not very efficient). The basic constraints are

- It must be possible to find all commit blocks written to disk since the last roll-forward start point.
- If a commit block is on disk, all previous commit blocks (starting at the roll-forward start point) are on disk, too.
- If a commit block is on disk, all the data it describes are on disk, too.

Note that there is no dependence between data writes belonging to different commit blocks, or between commit block writes and data writes belonging to later commit blocks.

We experimented with several ATA and SCSI disks (by performing synchronous user-level writes to the partition device), and found some interesting results:

Write caching in the drive can perform writes to disks out-of-order and has to be disabled if we want to satisfy the dependence constraints by write ordering.⁵ Without write caching, all disks we measured lose a disk revolution for each consecutive write (tagged command queuing won't help us when we have to wait for the acknowledgment of the

data write (and previous commit block write) before starting the current commit block write).

By placing the commit block at some distance (20KB in our experiments) behind the last data block, we can write the commit block in the same revolution and reduce the time until the commit is completed. This is useful in the case of frequent fsyncs. The blocks between the data blocks and the commit block can be used by the next write batch.

4 Current State and Further Work

Much of what we have described above is still unimplemented. LinLogFS currently performs all the usual file system functions except reclaiming space. It provides in-order semantics (even in the presence of fsync()). LinLogFS uses the roll-forward optimization, but does not use any change records yet.

The only component missing to make LinLogFS practically useful is the cleaner. Other features that are still missing and will be implemented in the future are: cloning (including writable, cloneable clones), change records for block pointers, pinning segments containing a file down (for LILO), fast update of network mirrors, efficient growing/shrinking/migrating the volume. We are also considering a version of LinLogFS as a low-level OBD driver (see www.lustre.org and Section 8).

5 Layering

5.1 Logging Block Device?

The difference between traditional and log-structured file systems is mainly in block writing. Most other parts of the file system (e.g., dealing with directories or permissions) are hardly affected.

So we first considered realizing the logging functionality as a block device driver. We decided against this, because:

- The block device driver interface is not sufficient to make a conventional file system log-structured without significant changes in the file system (atomicity for multiple block writes, crash recovery).
- A logging block device driver would interfere with other functionality implemented in the block device layer (software RAID).

⁵The disks we measured only wrote out-of-order if the same block was written twice (in some caching window), but there is no guarantee that other disks behave in the same way nor that we have seen all cases of out-of-order writing.

5.2 Logging Layer!

We decided to reuse most of the ext2 code. We defined a logging layer that sits below the rest of the file system. It is also possible to turn other file systems into log-structured file systems by adapting their code to work on top of our logging layer, but we have not done so yet.⁶

This separation into layers proved to be beneficial due to the usual advantages of modularization (e.g., separate testing and debugging).

The adaption of the ext2 filesystem turned out to be relatively straightforward because the ext2 implementation does a good job in encapsulating indirect block handling from the rest of the filesystem implementation. The interface provided by the logging layer is similar to the one offered by the buffer cache/block device driver thereby easing the adaption of the ext2 code.

Furthermore, we have added calls that bracket every filesystem operation to the ext2 filesystem layer. This ensures that dirty blocks do not get flushed to disk while a filesystem-modifying operation is taking place. This approach guarantees that the filesystem is always in a consistent state when a commit to disk is taking place.

6 Implementation Experiences

6.1 The Linux Kernel Framework...

Like many other implementations of Unix-like operating systems, Linux uses a uniform interface within the kernel (the *VFS Layer*) to access all the various filesystems it supports.

Usually local filesystems do not directly write to the block devices they reside on. They make use of the *buffer cache*, a unified cache that helps to avoid costly synchronous block device I/O operations in order to fulfill VFS requests.

Instead of doing synchronous block I/O, they rather mark blocks that are in the buffer cache as dirty (i.e., they need to be flushed to the device) when a

⁶Since logging changes the on-disk data structure, this would not be useful for file systems that are only used for compatibility, such as the MS-DOS file system. It would be useful for file systems that offer special features, performance or scalability advantages like XFS. We designed the logging layer and the on-disk structures to allow several adapted file systems to work on top of the logging layer at the same time (this is an extension of the cloning functionality).

VFS operation has caused a change in a block. It is then the task of the kupdate kernel thread to actually commit changes to the device asynchronously.

So a local filesystem can be seen as a filter that turns VFS calls into block read/write operations passed to the buffer cache.

The kupdate kernel thread will start writing dirty buffers back to the underlying block devices when one of the following conditions gets true:

- there are dirty blocks that have aged beyond a certain threshold,
- the operating system is running low on memory and needs to free up some RAM,
- there is an explicit request to update the block device, such as a sync() call.

6.2 ... And Changes Made to it

6.2.1 Extending The Buffer Cache

While this approach works fine for traditional file systems, such as ext2, it is not applicable for filesystems that want to ensure certain consistency guarantees for on-disk data structures. In order to be able to make such consistency guarantees, a filesystem implementation needs a more fine-grained control over the actual order in which writes are performed. However, this cannot be ensured by using the traditional buffer caching mechanisms for writes.⁷

LinLogFS also faces another difficulty using this approach. Since write operations are grouped into segments, the actual on-disk location of a block is not known until the block is actually about to be written out to disk. On the other hand, the buffer cache uses the device and the block number in order to locate a cached item.

Therefore, LinLogFS assigns block numbers past the physical end of the underlying device to unwritten dirty blocks. When a block is about to be written out, an *address fixup* is performed on the filesystem's meta data to reflect the final on-disk location. However, this cannot be handled using the buffer cache alone because some additional information is required in order to accomplish that task:

In order to be able to perform the address fixup, it is necessary to know the inode of the filesystem

⁷Of course it would be possible to bypass the buffer cache completely. But this is not desirable due to the performance impacts this approach will have.

Item	Modification
Filesystem Layer (new)	Based on ext2 Handles perms, dirs, etc... Performs address fixup on request
Logging Layer (new)	Lock dirty blocks in memory Trigger off address fixup Trigger off re-hashing of blocks after fixup
Buffer Cache	Added support for re-hashing blocks Prevent random writes Extended <i>buffer_head</i> for address fixup
Inode Cache	Added code for locating an inode in the cache

Figure 3: Overview of modifications to the Linux kernel

object the block belongs to and the logical offset within it. However, the Linux 2.2 buffer cache does not preserve this information.

Because of these obstacles, LinLogFS uses a slightly different approach when it comes to dealing with write operations: Instead of leaving the decision when to write which block to disk to the buffer cache, LinLogFS locks dirty blocks in the buffer cache. This prevents them from being flushed to disk by the *kupdate* daemon at free disposal. LinLogFS performs direct block I/O instead when enough blocks are dirty or a sync has been requested. This also allows the block address fixup to be performed just before the block is actually being written out to disk.

This approach of determining the address of a block upon the disk write could also be used to optimize other file systems, e.g., ext2: When the block is actually written to disk, usually its file is already closed. Therefore the file system knows the file's probable final size, and can allocate the files on the disk in a way that avoids file and free space fragmentation.

6.2.2 Implementing the Address Fixup

In order to be able to perform the address fixup on pending filesystem meta-data before a data block is written out to disk, we need to store some additional information about the block. Currently, this is done by extending the structure that is used to describe entries in the buffer cache *buffer_head* by a few additional members:⁸

⁸We are planning to reduce these additional members in the future. In general, we would suggest adding a general-purpose pointer member to this structure.

- *inum*: This member holds the inode number of the filesystem object this block belongs to.
- *log_blknum*: The offset (logical block number) of this block within the filesystem object it belongs to.
- *private_data*: A pointer to a data structure describing the LinLog filesystem. This eases the porting of existing filesystem implementations to LinLogFS's logging layer since this allows an interface similar to the one of the buffer cache to be used.⁹

When LinLogFS decides to write out dirty blocks the logging layer fixes up the block addresses in the *buffer_head* structs of all blocks that are about to be flushed to disk. It then issues a callback to the upper filesystem layer that is now supposed to adjust its meta-data information accordingly.

Since the buffer cache uses the on-device block number to hash entries, the respective blocks need to be re-hashed in the buffer cache after the address fixup has been performed. Therefore, a function has been added to the Linux buffer cache that allows to re-hash blocks in the buffer cache after the address fixup has been performed.

The address fixup is also complicated by the fact that there might actually be two copies of an inode in memory: One stemming from data blocks of the *.ifile* (the file containing all the inodes in LinLogFS) and another one being located in the inode cache. The inode cache in Linux is mainly used for storing a unified representation of a filesystem inode that has been obtained through the VFS layer. All processes wanting to access a certain file are associated with

⁹This allows us to simulate routines, such as *mark_buffer_dirty* or *bread*.

the representation of the file's inode in the inode cache.

However, the inode cache causes problems for LinLogFS, since it needs to make sure that the inode's copy in the inode cache remains consistent with the one in the .ifile's dirty buffers. Therefore, a call has been added to the Linux inode cache implementation that allows to obtain the current cached copy of an inode in the inode cache (if there is one at all). This allows us to locate a cached copy of an inode and to keep it consistent with the inode's actual state. We are currently considering to bypass the inode cache for filesystem-specific information in future versions of LinLogFS.

After the upper filesystem layer has completed the address fixup, the actual disk write is triggered by the logging layer by issuing an *ll_rw_blk* request to the underlying block device.

7 Performance

We compared several Linux file systems:

ext2 (Linux 2.2.13); this is the standard Linux file system, and represents traditional file systems. It performs all writes asynchronously. Ext2 does not give any consistency guarantees by ordering writes; instead, it relies on a sophisticated fsck program for crash recovery, but successful recovery is not guaranteed.

reiserfs (3.5.19 on Linux 2.2.14); this file system tries to scale well to dealing with large numbers of small files. Concerning recovery guarantees, this file system logs meta-data, so the directory tree will be preserved on crash recovery, but the data in it may not.

ext3 (0.0.2d on Linux 2.2.15pre15); this is the ext2 file system with journaling added; currently it logs both data and meta-data, which allows for very good consistency guarantees.¹⁰

linlogfs (on Linux 2.2.14); the version we measured did not contain a cleaner and does not keep track of segment usage as a version with a cleaner probably would; there are also several other changes planned that will affect performance. Concerning consistency guarantees, Linlogfs provides the in-order guarantee (see

¹⁰For higher performance at reduced safety meta-data-only logging is planned

Section 3.2), so crash recovery will restore both data and meta-data to the state a short time before the crash (ext3 probably provides a similar guarantee).

The hardware we have used for this performance comparison is based on a 450MHz AMD K6-III CPU, an Asus P5A mainboard (with ALI chipset), 128MB RAM and a 15GB IDE Harddisk (IBM-DJNA-351520).

For all benchmarks except one we disabled write caching on the hard disk with *hdparm -W0*. This is necessary for reiserfs, ext3, and LinLogFS, because with write caching enabled disks can perform the writes out-of-order in a way that would compromise the crash recovery guarantees these file systems make. Ext2 makes no crash recovery guarantees, but write caching must still be disabled to ensure that ext2 works correctly for *sync* etc. We have also run ext2 with write caching, because this is a common configuration (all IDE hard disks we know enable write caching by default).

Many factors influence file system performance, and it is probably impossible to reduce the wide range of usage patterns in real life to a small set of benchmarks that exercise all file systems in representative ways. So take the following with a grain of salt. The benchmarks we used were:

- Un-tarring the Linux 2.2.14 kernel sources, starting with a freshly-made file system; we ran this benchmark 50 times. Un-tarring is one of the few disk-intensive tasks we encounter in real life. This is a data writing benchmark.
- A single run of removing the 50 instances of the Linux kernel source trees generated by the previous benchmark. This is another one of the few disk-intensive tasks we encounter in real life; this benchmark mainly writes meta-data, in contrast to the un-tar benchmark.
- Tarring¹¹ the Linux 2.2.14 kernel (20 runs). This is a data reading benchmark.
- A single run of find, searching for "foo" (non-existent) in 20 Linux kernel source trees. This is a meta-data reading benchmark.

In addition to running these benchmarks on the various file systems, we also measured the limits of the hardware: We wrote 74MB (the same amount of

¹¹We used *tar c linux|cat >/dev/null*, because *tar* seems to optimize direct output to */dev/null*.

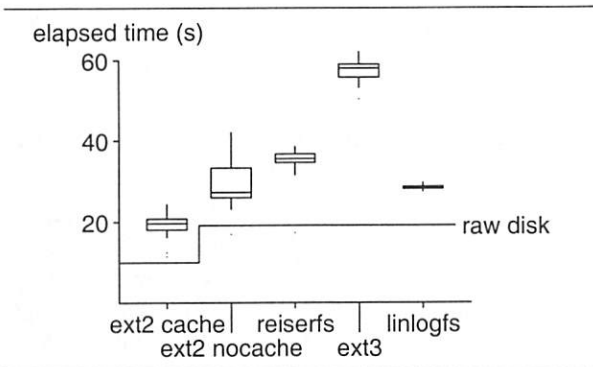


Figure 4: Un-tar timings

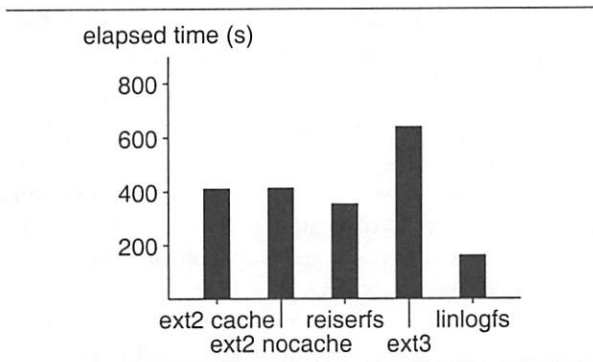


Figure 5: Rm timings

data as written by the un-tar benchmark) to the device with `dd if=/dev/zero of=/dev/hda7 ...` to measure the disk speed limit.

We ran the benchmarks using the default settings for the IDE chipset: no DMA (not supported for this chipset), no multiple sector access, 16-bit data transfers; with write caching disabled, enabling multiple-sector access and 32-bit data transfers make no difference for the elapsed time for direct device accesses (but reduced the CPU time somewhat). We also performed a few experiments with file systems and observed little difference between the two settings; we also believe that using DMA would not make much difference. The CPU load (as reported by `time`) in all the benchmarks except un-tar on ext2 cache was well below 50%, so the PIO probably did not interfere much with the file system.

With write caching enabled, raw writing becomes CPU-bound, and enabling these features reduces the elapsed time for writing 75MB to the device from 9.9s to 6.3s. So the ext2 cache results would be slightly better with the faster settings. Also, the read performance would be better with the faster settings.

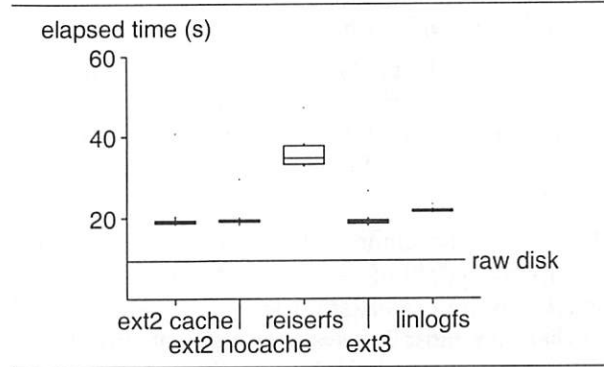


Figure 6: Tar timings

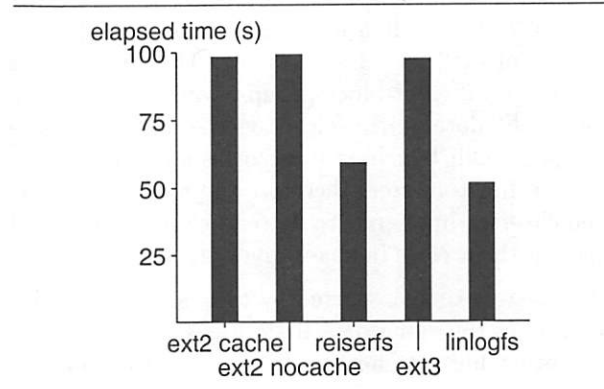


Figure 7: Find timings

The results are shown in Fig. 4, Fig. 5, Fig. 6, and Fig. 7. For the un-tar and tar benchmarks, we display the result as a box-plot showing the median, the quartiles, extreme values, and outliers.

In the local un-tar benchmark, we see that turning off write caching reduces raw disk write performance almost by a factor of 2, but ext2 write performance only by a factor of about 1.5. Ext2 utilizes 2/3 of the write bandwidth of the disk without caching, which is hard to beat. Between ext2, reiserfs and ext3 we see that file systems providing more consistency guarantees perform somewhat slower. LinLogFS breaks this trend, performing as well as ext2 nocache on this benchmark; of course, once cleaning comes into play the performance will suffer. Similarly, the performance of the other file systems on a fuller, aged file system might be worse, too. The outliers below the *raw disk* line that you see for ext2 nocache and reiserfs are probably due to write caching by the file system before memory becomes full.

In the rm benchmark LinLogFS wins, probably because it does not do any bookkeeping of segment usage or free blocks yet; you can expect some slow-

down in LinLogFS rm performance in the future.

In the tar and find benchmarks, we see that ext2 cache, ext2 nocache, and ext3 perform the same, because neither caching nor journaling plays a role when reading (i.e., for reading ext3 is the same as ext2).

For the tar benchmark, LinLogFS performs worse than ext2 by about 15%, probably because of the current write organization within a segment. The outliers for most file systems are probably due to interference with the delayed writes of the un-tars that immediately precede the tar benchmark.

For the find timings, LinLogFS and reiserfs are significantly faster than ext2/ext3. The lower performance of ext2 may be due to the way it allocates inodes to different block groups, whereas in a clean LinLogFS data written close together in time is close in space. Our benchmark performs a find on an untarred directory tree; therefore the meta-data written close in time tends to be read close in time, and having them close in space gives an advantage.

The system times reported by time show LinLogFS as taking the same or a little less CPU time than the other file systems on all benchmarks; however, take this with a grain of salt, because we saw some strange results in our raw disk measurements, so the reported system time for I/O-intensive benchmarks is not necessarily accurate.

8 Related Work

In many respects LinLogFS is similar to the Sprite LFS [RO92] and especially the BSD LFS [SBMS93]. The most significant difference is that LinLogFS is designed to allow efficient cloning of the file system. One important difference from Sprite LFS is the logical separation of commits and segment summaries; Sprite LFS views segment boundaries as commits and requires additional complexity like the *directory operation log* to work around the resulting problems; BSD LFS logically separates commits from segments (look for *segment batching* in [SBMS93]). BSD LFS does not use change records, Sprite LFS uses the segment summaries as change records for block pointer update optimization.

Another interesting aspect is that the original motivation for log-structured file systems (in particular, Sprite LFS) was performance [OD89], while our motivation is mainly functionality¹² (the performance

of ext2 in the usual case is satisfactory). For performance evaluations of log-structured file systems see [SSB⁺95, MRC⁺97].

Cleaning heuristics and performance are discussed extensively in [RO92, BHS95].

The Spirallog file system [WBW96] for VMS is a log-structured file system with a number of interesting differences from the file systems discussed above. One of the differences to LinLogFS is its approach to backups [GBD96]: Spirallog just physically backs up all live segments; for incremental backups it backs up all live segments written in the corresponding range of time. In contrast, in LinLogFS you will create a logical read-only clone of the file system, and use your favourite logical backup tool (e.g., tar) to make the backup.

Network Appliance's WAFL file system [HLM94] differs from LinLogFS mainly in using free block maps instead of segments and by not performing roll-forward of on-disk structures (i.e., every commit needs a write to the superblock); WAFL is supported by an NVRAM buffer, mitigating the worst-case performance impacts of these decisions. WAFL's allocation scheme trades additional seek times for the elimination of the cleaning overhead; moreover, creating snapshots¹³ is significantly more expensive (tens of seconds). Concerning implementation complexity, WAFL needs no cleaner, but probably incurs some additional complexity in the block allocator.

There are two other Linux log-structured file system projects we know of, by Cornelius Cook, and by Adam Richter, but to our knowledge they stalled at some early stage.

Journaling (e.g., ext3 [Twe98]) augments conventional file systems for fast crash recovery by writing data first to a log and later to the home location; after a crash the log is replayed to get a consistent on-disk state. Space in the log is reclaimed by completing writes to the home location. Most journaling file systems (e.g., BFS [Gia99], Calaveras [VGT95], Episode [CAK⁺92], IBM's JFS, reiserfs) only log meta-data writes (for performance reasons), and therefore only guarantee meta-data consistency. Journaling file systems can provide fast crash recovery and (with data logging) in-order write semantics and relatively good performance for synchronous writes. But they do not easily support cloning,

but this is not the kind of performance that most papers focus on.

¹³User-visible snapshots; WAFL also uses the term *snapshot* for commits; commits are not that expensive.

¹²You can see crash recovery speed as a performance issue,

snapshots or other features that are easily available in log-structured file systems. Still, Episode [CAK⁺92] supports read-only clones, using block-level copy-on-write.

Soft updates [MG99] enhance BSD's FFS by asynchronously writing in an order that ensures that the file system can be safely mounted after a crash without being checked. They make FFS's writes and crash recovery faster. McKusick and Ganger [MG99] also show how to do snapshots in a conventional file system. While they address our main motivations for developing LinLogFS, their software does not work under Linux, and we believe that their solution is more complex to implement than a log-structured file system.

The logical disk [dJKH93] provides an interface that allows a log-structured layer below and an adapted conventional file system on top. The logical disk translates between logical and physical blocks, requiring considerable memory and long startup times. In contrast, our logging layer uses physical block addresses and only affects writes.

The Linux logical volume manager LVM 0.8final supports multiple snapshot logical volumes per original logical volume. This allows making consistent backups for any Linux file system.

The object based disk (OBD) part of the Lustre project (www.lustre.org) divides the file system into a hierarchical layer on top, and a flat layer of files at the bottom, like the layering presented in Section 2. OBD allows inserting *logical object drivers* between these layers. In particular, there is a snapshot driver that uses copy-on-write on the file level. The interface between the OBD layers is higher-level than the interface between the LinLogFS layers. We are considering to have a derivative of LinLogFS as the lower layer in OBD (currently a derivative of ext2fs is used).

9 Conclusion

Log-structured file systems offer a number of benefits; one of the little-known benefits is in-order write semantics, which makes application data safer against OS crashes. LinLogFS is a log-structured file system for Linux. It is implemented by adding a log-structured layer between an adapted ext2 file system and the buffer cache. LinLogFS defers the assignment of block addresses to data blocks until just before the actual device write; it assigns temporary block addresses to dirty data blocks at first,

and fixes the addresses just before writing.

The current version of LinLogFS works on Linux 2.2 (versions for Linux 2.0 are available). You can get it through www.complang.tuwien.ac.at/czezatke/lfs.html.

Acknowledgments

Stephen Tweedie (our shepherd), Ulrich Neumerkel, Manfred Brockhaus, and the referees provided helpful comments on earlier versions of this paper.

References

- [BHS95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Usenix Annual Technical Conference*, pages 277–288, 1995.
- [CAK⁺92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *Usenix Conference*, pages 43–60, Winter 1992.
- [dJKH93] Wiebren de Jonge, M. Frans Kashoek, and Wilson C. Hsieh. Logical disk: A simple new approach to improving file system performance. Technical Report LCS/TR-566, MIT, 1993. A paper on the same topic appeared at SOSP '93.
- [GBD96] Russel J. Green, Alasdair C. Baird, and J. Christopher Davies. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, 8(2):32–45, 1996.
- [Gia99] Dominic Giampaolo. *Practical File System Design*. Morgan Kaufmann, 1999.
- [HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Usenix Conference*, Winter 1994.
- [MG99] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem.

In *FREENIX Track, USENIX Annual Technical Conference*, pages 1–17, 1999.

- [MRC⁺97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Sixteenth ACM Symposium on Operating System Principles (SOSP '97)*, 1997.
- [OD89] John Ousterhout and Fred Douglass. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, January 1989.
- [RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [SBMS93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Usenix Conference*, pages 307–326, Winter 1993.
- [SSB⁺95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Usenix Annual Technical Conference*, 1995.
- [Twe98] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.
- [VGT95] Uresh Vahalia, Cary G. Gray, and Dennis Ting. Metadata logging in an nfs server. In *Usenix Annual Technical Conference*, pages 265–276, 1995.
- [WBW96] Christopher Whitaker, J. Stuart Bayley, and Rod D. W. Widdowson. Design of the server for the Spirallog file system. *Digital Technical Journal*, 8(2):15–31, 1996.

Unix file system extensions in the GNOME environment

Ettore Perazzoli

Helix Code, Inc.

ettore@helixcode.com, <http://primates.helixcode.com/~ettore/>

1 Introduction

This paper explains how the GNOME [GNOME] project is extending the functionality of the Unix file system for use in both the desktop and applications, by using a user-level library called the GNOME Virtual File System.

There are various reasons for extending the Unix file system and its API, as explained in the following sections.

1.1 Uniform file access

In a modern desktop environment, users have to deal with files that are available in different ways. For example, files can be remotely available from a Web or FTP site. Sometimes they are stored locally, but are contained in tar or zip files. Sometimes they are stored in a compressed form.

Each of these cases usually requires the user to use a different tool; using different tools means that users have to learn the user interface for each of them, and have to manually make these tools talk to the applications, for example by using temporary files.

To avoid this problem, there should be a global file system namespace that all applications can use to access these different kinds of files. Users should not need to install and use different programs to access files available through different methods, and all the applications should be able to access this global file system namespace.

So basically we need a consistent API that all the applications can use to access these

files in a simple way.

1.2 Representing special non-file objects

Current Unix desktop environments lack a unified file system -like representation of the system resources. In the Windows operating system, users can access all the system resources from the “My Computer” folder which acts as the root of the operating system shell’s “namespace”. This means that the control panel, the printer’s spool, the trash-can, the removeable devices and so on are all accessible through the same simple “point-and-click” mechanism within the same application (Explorer.exe). Unlike the current Unix desktop environments, users don’t need to deal with different applications for the various objects that are in their computer: they can just use the desktop shell.

To implement similiar functionality on Unix, we would need to have an extensible mechanism for providing the contents to these virtual folders and implementing operations that can be performed on them.

1.3 Asynchronous operation

On Unix, there is no API to let applications do fully asynchronous I/O in an easy and portable way.

The reason why this is so important is that GUI applications need to be responsive all the time. If a GUI application is blocked during a long synchronous file operation, the user cannot stop the operation, nor perform any other task with the same application. This

is especially important for a file manager, which needs to be able to perform multiple I/O tasks in parallel without taking control away from the user.

One of the ways of dealing with asynchronous I/O, i.e. using the `select()` system call, does not work with all the operations; for example, there is no way to make an asynchronous `gethostbyname()`, `open()` or `stat()` without complicated hacks.

Unix programmers could also use POSIX threads for performing asynchronous operations, but thread programming is very error-prone and can easily lead to problems. Writing a threaded application requires more programming skills writing a non-threaded one, and in the free software world it is very important to make things easy for programmers.

Moreover, POSIX threads are not fully portable across different platforms, as not all implementations are reliable and efficient.

Finally, although a POSIX API for asynchronous I/O exist, it is not implemented on all systems and does not work with all system calls. For example, you cannot execute the `open()` system call in an asynchronous fashion. Moreover, this API does not fit nicely in the event-based model of GUI systems; consequently, it is not suitable for rapid development of GUI applications.

What we need for asynchronous operation is an easy to use API that hides most of the details from the programmer and integrates nicely with the existing GUI toolkits. The API should give a better abstraction for doing async I/O, in the most simple way possible.

2 Existing User Space Virtual File Systems

In the past, there have been other attempts to implement extensions to the Unix file system in the free software world, none of which completely fulfilled the needs of the GNOME project.

2.1 The GNU Midnight Commander

The GNU Midnight Commander is the file manager of choice for the GNOME project at the time of writing. It implements a user-space Virtual File System library that solves the problem of accessing files within archive files or on remote Internet sites through an extended URI system which is explained in greater detail in section 4.

Although the VFS library source code could be made independent of the GNU Midnight Commander and thus its functionality could be used by all the applications, it suffers a few design problems:

- It does not support asynchronous operation: if you perform an operation on a remote system, the Midnight Commander will be blocked until the operation is complete, and the user is not even able to stop the operation.
- The library is hidden under a Unix API that is not very extensible.

The latter problem is the most important one: as a Virtual File System has to deal with kinds of files that can be quite different from the standard Unix files, it needs some API extensions to support this functionality. Moreover, it needs some API extensions to deal with asynchronous operations. Unfortunately, it is not possible to add functionality to the API in a clean way without breaking Unix compatibility, so this makes the GNU Midnight Commander's Virtual File System unsuitable as a generic library for GNOME.

2.2 KDE's kio

The K Desktop Environment [KDE] provides a completely asynchronous virtual file system library called `kio`.

`kio` uses the same extended URI syntax that the GNU Midnight Commander uses; but unlike the Midnight Commander, all the

file operations are performed through external helper processes (kioslaves) in an asynchronous fashion. The helper process communicates with the master process using Unix domain sockets and custom protocols; the library performs encoding/decoding of such protocols automatically and notifies the programmer using the Qt [Qt] signal system.

Although this system is closer to the requirements that we have listed at the beginning of this paper, it is still suboptimal, for the following reasons:

- Operations are very simple: the API allows the programmer to download/upload a whole file and perform some simple file operations, but it does not provide all the versatility of the Unix API.
- Each file operation requires an external process: kio does not take advantage of threads.
- There is no API to perform operations in a synchronous fashion, so the programmer needs to spawn a process, request an operation and then wait for the result from it.

3 The GNOME Virtual File System

The solution we gave to these issues in the GNOME project is in the form of a new library, designed from scratch to meet all the requirements. This library (the GNOME Virtual File System, or GNOME VFS for short) takes advantage of the existing GNOME development libraries, such as GLIB and GTK+ [GTK].

The following design ideas were kept in mind while implementing the GNOME VFS:

- Like the rest of GNOME, the API should be C-based and follow the standard GNOME programming style.

- The implementation should be portable, so that making GNOME VFS a core component of GNOME would not restrict the availability of GNOME on various Unix platforms.
- Using GNOME VFS should not make the programmer's life harder.
- The asynchronous API should be simple to use, and be nicely integrated with the standard GLIB main loop that is central to GNOME applications. (The GLIB main loop is the main event-handling loop in GTK+ and GNOME applications. The X main loop is nicely wrapped by the GLIB loop.)
- Adding new access methods should be possible, and programmers should not need to care about too many of the details, such as asynchronous behavior. (By "access method", we mean the implementation of a protocol, of a file format, or any other way to retrieve, create or modify a GNOME VFS file. For example, there should be an HTTP access method, a ZIP file access method and so on.)

4 Extended URIs

The GNOME VFS uses an extension of the traditional web Uniform Resource Identifiers (URIs) to access files, instead of the standard Unix file paths. This enables us to access different kinds of resources in a way that is both generic and easy to understand for users.

GNOME VFS extended URIs are reminiscent of the GNU Midnight Commander's URI syntax, and consequently use the # character to stack access methods on top of each other.

In the simplest form, a GNOME VFS URI looks exactly like a normal Web URI, that is:

```
method://user:password
@some.host/path/to/file
```

As in normal Web URIs, user and password

are optional; in that case, the @ character will be omitted as well.

For example, the URI

```
http://www.gnome.org/index.html
```

will refer to the file `index.html` from the host `www.gnome.org` through the `http` access method, which is an implementation of the HTTP protocol.

The access method for the local file system is called `file`, so, for example, the file `/etc/passwd` is accessed through the URI

```
file:/etc/passwd
```

Unlike normal Web URIs, though, GNOME VFS URIs let you “stack” access methods on top of each other. GNOME VFS, in fact, supports two kinds of access methods: “`oplevel`” access methods, that access files directly, and “`archive`” access methods, that access files within other files.

For example, there is a `zip` access method that lets you access files contained in a `.zip` archive file: the `.zip` access needs a “parent” access method to access the archive in which the file is contained.

Stacking is achieved in GNOME VFS URIs by using the special character `#`. The generic syntax is:

```
uri#method[/sub_uri]
```

When this syntax is used, it specifies that `sub_uri` must be accessed within the file available through `uri` using the specified method.

For example, imagine you have a `foo.zip` archive containing a file called `bar.c`. Also suppose that `bar.c` is contained in a directory called `baz` within `foo.zip`, and that `foo.zip` is in your home directory `/home/joe`. The URI to specify `foo.zip` is:

```
file:/home/joe/foo.zip#zip/baz/bar.c
```

If the file was available through FTP instead, the URI would be something like:

```
ftp://joe:passwd@ftp.site.net  
/home/joe/foo.zip#zip/baz/bar.c
```

Some access methods don’t require a sub-path; for example, this is the case with the `gzip` access method that can be used to read and create compressed files in `.gzip` format.

If you wanted to read the contents of a compressed `foo.gz` file in your home directory, you would simply have to specify the following URI:

```
file:/home/joe/foo.gz{tt \#}gzip
```

By combining the `gzip` access method with the `tar` access method, it is possible to access files contained in `tar.gz` archives:

```
file:/home/ettore/download  
/gnome-vfs-0.1.tar.gz#gzip  
#tar/gnome-vfs-0.1/AUTHORS
```

There is no limitation in the amount of stacking that can be performed. Every access method in GNOME VFS is implemented as a dynamically loaded plug-in, as described in section 10: anyone can extend the VFS with new access modules.

5 GNOME VFS API basics

Almost all the standard Unix file operations have counterparts in GNOME VFS. In the following sections, we will give a brief overview of the GNOME VFS API.

5.1 The GnomeVFSURI object

A GNOME VFS URI is represented by a special GNOME VFS object called `GnomeVFSURI`. `GnomeVFSURI` objects are the

preferred way to specify files: users of the library can use `GnomeVFSURIs` to store and manipulate URIs, and the interface between the library and its plug-ins uses `GnomeVFSURI` objects.

A `GnomeVFSURI` object is created by using the call

```
GnomeVFSURI *
gnome_vfs_uri_new (const char *s)
```

`GnomeVFSURIs` also have a reference count that can be controlled by using the calls:

```
void gnome_vfs_uri_ref (GnomeVFSURI *uri)

void gnome_vfs_uri_unref (GnomeVFSURI *uri)
```

A `GnomeVFSURI` can be also converted into a printable string by using the following call:

```
char *gnome_vfs_uri_to_string
(const GnomeVFSURI *uri,
 GnomeVFSURIShowOptions hopt)
```

It is also possible to extract the host, user name and password information from it, as well as compare and combine them. Virtually any path operation that the programmer might need is supported directly through the `GnomeVFSURI` API.

5.2 The `GnomeVFSResult` enumeration

All the GNOME VFS operations return a result value of type `GnomeVFSResult` that represents the result of the operation. `GnomeVFSURI` is a numeric enumeration:

```
enum _GnomeVFSResult {
    GNOME_VFS_OK,
    GNOME_VFS_ERROR_NOTFOUND,
    GNOME_VFS_ERROR_GENERIC,
    GNOME_VFS_ERROR_INTERNAL,
    GNOME_VFS_ERROR_BADPARAMS,
    GNOME_VFS_ERROR_NOTSUPPORTED,
    GNOME_VFS_ERROR_IO,
```

```
    GNOME_VFS_ERROR_CORRUPTEDDATA,
    /* ... */
    GNOME_VFS_NUM_ERRORS
};
typedef enum _GnomeVFSResult
    GnomeVFSResult;
```

Just like the Unix `errno` variable, you can get a string description from a `GnomeVFSURI` value by using the following function:

```
const gchar *gnome_vfs_result_to_string
(GnomeVFSResult result)
```

6 Synchronous API

In GNOME VFS, both a synchronous and asynchronous API call exist most file operations. The synchronous versions work like normal Unix calls: they perform the operation, then return and report success/failure using a `GnomeVFSResult` value.

6.1 The `GnomeVFSHandle` object

As in the Unix API, files need to be “open” before being ready to be read or written. But while the Unix API returns a simple integer to represent a “file handle”, the GNOME VFS API provides a special object type for that, called `GnomeVFSHandle`.

`GnomeVFSHandle` objects are created using the “open” or “create” calls

```
GnomeVFSResult
gnome_vfs_open_uri
(GnomeVFSHandle **handle return,
 GnomeVFSURI *uri,
 GnomeVFSOpenMode open mode)
```

```
GnomeVFSResult
gnome_vfs_create_uri
(GnomeVFSHandle **handle return,
 GnomeVFSURI *uri,
 GnomeVFSOpenMode open mode,
 gboolean exclusive,
 guint perm)
```

and destroyed by using the “close” call:

```
GnomeVFSResult
gnome_vfs_close (GnomeVFSHandle *handle)
```

6.2 Synchronous I/O Example

Here is a simple example demonstrating synchronous operation in GNOME VFS. This subroutine will read a file and output its contents to stdout. The code should be rather self-explanatory.

```
gboolean vfs_cat (const char *uri)
{
    GnomeVFSURI *vfs_uri;
    GnomeVFSHandle *handle;
    GnomeVFSResult result;

    vfs_uri = gnome_vfs_uri_new (uri);
    if (vfs_uri == NULL) {
        printf ("'%s' is not a valid URI.\n",
            uri);
        return FALSE;
    }

    result = gnome_vfs_open_uri
        (&handle, vfs_uri, GNOME_VFS_OPEN_READ);

    if (result != GNOME_VFS_OK) {
        printf ("Error opening '%s': %s\n",
            uri,
            gnome_vfs_result_to_string (result));
        return FALSE;
    }

    while (1) {
        GnomeVFSFileSize bytes_read;
        GnomeVFSFileSize i;
        char buffer[4096];

        result = gnome_vfs_read
            (handle,
             buffer,
             sizeof (buffer),
             &bytes_read);
        if (result != GNOME_VFS_OK) {
            printf ("%s: %s\n", uri,
                gnome_vfs_result_to_string
                    (result));
            return FALSE;
        }

        if (bytes_read == 0)
            break;
        for (i = 0; i < bytes_read; i++)
            putchar (buffer[i]);
    }

    gnome_vfs_close (handle);
    gnome_vfs_uri_unref (vfs_uri);
    return TRUE;
}
```

7 Asynchronous API

In the asynchronous API calls, instead, the operation requested is started in a separate thread of execution and control returns to the caller immediately. The caller will have to specify a callback function that will be called when the operation is completed. If the operation is long to perform (such as a "copy" operation), the callback might be called more than once to report progress.

All the synchronous API calls have asynchronous counterparts; their name is the same as the synchronous one, but use the `gnome_vfs_async` prefix, instead of just `gnome_vfs`.

Callbacks for asynchronous operations are triggered in the normal GLib/GTK+ event loop. This means that the application will be able to handle GUI events and GNOME VFS events simultaneously and transparently. The programmer just needs to set up the callback functions and make sure the event loop is running all the time. This makes usage of GNOME VFS in GUI applications very convenient and easy to use.

7.1 The GnomeVFSAsyncHandle object

If an asynchronous operation is started successfully, the caller is given back a `GnomeVFSAsyncHandle` object that can be used to stop the operation after it has been started, by using the following call:

```
GnomeVFSResult gnome_vfs_async_cancel
(GnomeVFSAsyncHandle *handle)
```

In the case of file "open" and "create" operations, the `GnomeVFSAsyncHandle` object can also be used to request read/write operations on the file after it has been opened or created.

7.2 Opening a file as a GLib I/O channel

A common application case is when the program needs to get data from a file stream as soon as it becomes available from the transport layer. In the case of Unix, this is achieved through the `select()` system call.

GLib abstracts this mechanism by merging it with the event handling loop, through objects known as `GIOChannels`. With `GIOChannels`, it is possible to attach callbacks to a file descriptor, and have functions called as soon as data becomes available on it.

Special GNOME VFS API functions allow the programmer to open and read (or write) a file through a `GIOChannel`. For example, a file can be opened by using the following function:

```
GnomeVFSResult
gnome_vfs_open_as_channel
(GnomeVFSAsyncHandle **handle return,
 const gchar *text uri,
 GnomeVFSOpenMode open_mode,
 guint advised_block_size,
 GnomeVFSAsyncOpenAsChannelCallback
 callback,
 gpointer closure);
```

Notice that, unlike the normal Unix `select()` call, this is reliable with local files too: the application will never be blocked after reading from a file for which the “data available” callback has been called.

This is possible because of the GNOME VFS asynchronous engine described in section 11.

7.3 Asynchronous API example

The following function reads a file asynchronously, with the output sent to `stdout`.

```
#define BUFFER_SIZE 4096

/* This is the callback that
   will be called whenever
   something happens on the
```

```
I/O channel associated
with the file. */
```

```
static gboolean
io_channel_callback
(GIOChannel *source,
 GIOCondition condition,
 gpointer data)
{
    char buffer[BUFFER_SIZE + 1];
    unsigned int bytes_read;
    unsigned int i;

    if (condition & G_IO_IN) {
        /* Data is available. */
        g_io_channel_read
            (source, buffer,
             sizeof (buffer),
             &bytes_read);

        for (i = 0; i < bytes_read; i++)
            putchar (buffer[i]);

        fflush (stdout);
    }

    /* An error happened while reading
       the file. */

    if (condition & G_IO_NVAL)
        return FALSE;

    /* We have reached the end of the
       file. */

    if (condition & G_IO_HUP) {
        g_io_channel_close (source);
        return FALSE;
    }

    /* Returning TRUE will make sure
       the callback remains associated
       to the channel. */

    return TRUE;
}

static void
open_callback (GnomeVFSAsyncHandle *handle,
               GIOChannel *channel,
               GnomeVFSResult result,
               gpointer data)
{
    if (result != GNOME_VFS_OK) {
        printf ("Error: %s.\n",
                gnome_vfs_result_to_string
                    (result));

        return;
    }
    printf ("Open: '%s'.\n",
            (char *) data);
    g_io_add_watch_full
        (channel, G_PRIORITY_HIGH,
         G_IO_IN | G_IO_NVAL | G_IO_HUP,
         io_channel_callback,
         handle, NULL);
}
```

```

void
start_cat (const char *uri)
{
    GnomeVFSAsyncHandle *handle;

    /* Start the 'read' operation. */
    gnome_vfs_async_open_as_channel
        (&handle, uri, GNOME_VFS_OPEN_READ,
         BUFFER_SIZE, open_callback, "data");
}

```

8 File attributes

GNOME VFS also provides a `GnomeVFSFileInfo` object that works as an extension of `struct stat` in Unix. In addition to the standard attributes that `struct stat` provides, `GnomeVFSFileInfo` also gives access to:

- MIME type information. GNOME VFS is able to take advantage of plug-ins for which the MIME type is part of the information that the access method provides. For example, in the case of the HTTP back-end, GNOME VFS can provide the MIME type as returned by the HTTP server.
- Metadata. Arbitrary value/key pairs can be associated with files, for generic purposes. For example, you can use an “icon” value for specifying a file’s icon, or a “description” value for a verbose description of the file.

As some kind of information might not be supported by certain plug-ins (for example, it is not possible to know the number of physical blocks occupied by a file via HTTP), `GnomeVFSFileInfo` provides a bitmask specifying which fields are actually valid and which are not.

GNOME VFS also provides a simple API for loading directory information in a progressive way, calling an asynchronous callback as data is copied into memory. This functionality is particularly useful for a file manager, as the directory view can be updated without blocking the user interface and thus giving the user effective visual feedback of what

is going on, even with those slow back-ends for which reading a directory is an expensive operation (such as a tar file access method, that requires a sequential scan of the whole .tar file).

9 File transfer support

In GNOME VFS there is a special API call for copying and moving files, while providing the caller with progress information while the operation is being performed. GNOME VFS is able to automatically optimize the case in which a file is moved through two different locations on the same physical file system, by using the information provided by the source and destination plug-ins.

When this API call is used, all the file transfer is performed in a separate thread or process, which dispatches the progress information to the main thread/process using the same mechanism that is used by the other asynchronous calls.

In order to reduce the impact of dispatching progress information across process/thread boundaries, the actual calls take place periodically, at a rate specified by the programmer. In the case of a file manager, for example, this will be done only a few times in a second, just enough to make sure the GUI is updated the way the user would expect.

10 Implementation of access plug-ins

As explained in 4, access methods are implemented as dynamically loaded plug-ins. Even the file plug-in that accesses local files is implemented as a plug-in.

Dynamic loading of plug-ins happens during `GnomeVFSURI` parsing (see section 5.1): the URI is split into its #-separated subparts, the library looks up the access method names (such as file, http and so on), and tries to locate the corresponding plug-in library, by us-

ing both a system-wide and a user-wide configuration file called `gnome-vfs.conf`. If the library is located, it gets linked in dynamically; otherwise, a return code is returned, reporting that the URI is invalid.

Every access method module must provide an initialization function; the initialization function returns a simple vtable containing pointers to the implementations of the GNOME VFS operations for that access method. Pointers to these vtables are stored directly into the `GnomeVFSURI` object, ready for the first operation to be invoked on it.

At the time of writing, the following modules have been implemented:

- A `bzip2` module, for files that are compressed with the `bzip` program.
- A `gzip` module, for files that are compressed with the `gzip` program.
- An `ftp` method for accessing remote sites through the Internet FTP protocol.
- An `http` method for accessing remote sites through the HTTP 1.1 protocol. This module also supports WebDAV.
- An `extfs` module, supporting Midnight Commander's generic archive file support based on simple shell scripts. This multipurpose module makes GNOME VFS able to deal with `zip`, `zoo`, `rpm`, `deb`, `arj` and other formats.
- A `gconf` protocol to access the new GNOME configuration database, called `GConf`.

Other modules, including a `tar` one, are being developed.

While this is not supported at the time of writing yet, there are also plans to support CORBA-based access method plug-ins, possibly using the Bonobo component model. [CORBA]

This will have important consequences:

- It will become possible to write plug-ins in any CORBA-aware language. For example, you could have plug-ins written in Perl, Python, or other scripting languages for which CORBA support exists. This will make it extremely simple for people to come up with their own special scripts for special directories or file systems.
- It will become possible to make GNOME VFS see file systems that are implemented in a different process. The external process might be running all the time, and would be contacted through CORBA. For example, you see the list of active print jobs in the print spooler as a normal GNOME VFS directory.

11 Implementation of asynchronous operations

To make operations totally asynchronous, we need to be able to perform them independently of the code that wants them to be executed. This can be done using either external helper processes or helper threads.

The thread-based solution has a number of advantages:

- It requires less memory to work.
- It is faster, as no data needs to be transferred from the helper to the master.

Unfortunately, it is not fully portable, as many systems don't have a suitable thread support. For this reason, GNOME VFS implements asynchronous operation in both ways. This is done by splitting the library in two parts: the basic file access library, and the asynchronous wrapper library. While there is one single version of the former, there are two versions of the latter: one that is based on POSIX threads, and one that uses external processes.

At run time, GNOME VFS applications are dynamically linked to either of the two

wrapper libraries. This makes it possible to use either method without changes in the source code. In the case of helper processes, the GNOME Virtual File System uses CORBA to communicate with them.

Using CORBA has the advantage of not requiring the creation of a custom inter-process communication protocol; moreover, adding new operations is very simple (you just need to extend the IDL). GNOME comes with its own ORB by default, so this does not add any further constraints to the applications willing to use GNOME VFS.

12 GNOME VFS and GNOME

GNOME VFS is a core component of the upcoming GNOME 2.0 platform. When GNOME 2.0 is released, GNOME VFS will be available for all applications (GUI or non-GUI) to use. GNOME VFS is also a central component of the new GNOME file manager and shell, which is called Nautilus [Nautilus] and is currently being developed by Eazel, Inc. [Eazel]. Nautilus makes extensive use of asynchronous operations and file system abstractions to improve the usability of the GNOME desktop. The first public release of Nautilus is expected in Summer 2000.

13 Availability

GNOME VFS is available from the GNOME FTP repository:

```
ftp://ftp.gnome.org
/pub/GNOME/unstable/sources/gnome-vfs
```

It is also possible to access the source code through the GNOME anonymous CVS system `anoncvs.gnome.org`, as explained at the URL

```
http://developer.gnome.org/tools/cvs.html
```

The name of the module is `gnome-vfs`.

The GNOME CVS has a web front-end available at the URL

```
http://cvs.gnome.org
```

References

[GNOME] The GNOME home page,
<http://www.gnome.org>

[HelixCode] Helix Code, Inc.
<http://www.helixcode.com>

[KDE] The KDE home page,
<http://www.kde.org>

[Qt] Qt
<http://www.troll.no>

[GTK] GTK+
<http://www.gtk.org>

[Nautilus] Nautilus
<http://nautilus.eazel.com>

[CORBA] CORBA
<http://www.corba.org>

[Eazel] Eazel, Inc.
<http://www.eazel.com>

Protocol Independence Using the Sockets API

Craig Metz

Department of Computer Science

University of Virginia

Charlottesville, VA 22904

`cmetz@inner.net`

Abstract

The BSD sockets API provides abstractions and other features that help applications be protocol-independent. Unfortunately, not all of the API is abstract and generic, and many programs do not use the APIs in a protocol-independent way. This means that most network programs, in practice, only work with one layered set of communications protocols – usually TCP over IP. This hinders compatibility with older protocols and deployment of new ones, and is making IP a victim of its own success.

During the course of next-generation IP development, implementors worked to convert protocol-dependent applications into protocol-independent applications. Along the way, they defined new interfaces to fix some problems and they found a number of usage problems that lead to protocol dependencies.

This paper explains many of the problems encountered, using examples from freely available software, and how to solve them. It also explains many of the new protocol-independent interfaces.

1 Introduction

The single most painful lesson learned by implementors of next generation IP proposals (such as IPv6) was how deeply most network programs are dependent on the network protocol that they were originally written to use. The widespread success of the IP Internet has put it into the position of being the only network protocol that matters for most network applications, and so there is currently little incentive to support anything else.

Even today, this is a problem. There are other network protocols that are in use today – such as Appletalk, ATM, AX.25, DECnet, Frame Relay, IPX, and OSI – and few applications actually support them. This is also a serious problem for the future, as any research into new network protocols is greatly constrained by the lesson of IPv6: that anything not IP

will not be supported by the applications people want to use, and that anything that is not supported by existing applications will encounter great difficulty gaining acceptance.

The core of the BSD sockets API, especially the actual system calls, is not tied to any particular protocol. The problems fall in two major categories: supporting APIs that are protocol-dependent, and poor programming practices that are common. There have been great advances made in fixing the networking APIs in the system libraries though the efforts of the IETF's IPng working group[1] and the IEEE's POSIX p1003.1g (networking API) working group. New library functions, data structures, and pre-processor symbols together allow addresses and other network properties to be treated as variable-length abstract objects whose internal format can be changed without the application's involvement. But there is still a serious problem of programmer education, which in turn requires good documentation of the problems and how to solve them. To date, this documentation still does not exist.

In the NRL IPv6 implementation[2], standard networking applications from BSD and from the Internet were taken and modified to support protocol independence. For most applications, this was straightforward once we had done a few applications and knew what to look for. The end result was not only a conversion to allow the applications to support almost any protocol, but also a significant cleanup of the applications' code.

In this paper, I will first discuss the problems that need to be solved to make programs protocol-independent: what is wrong, why it is wrong, and how it can be done right. I will then discuss the new protocol-independent API functions and compare them with older BSD networking API functions in light of the problems that need to be solved. I will also present in more detail some additional functions that we found necessary to solve certain problems that have not yet been standardized. Specific examples from

freely available networking programs will be used.

2 Protocol Independence Problems

Fundamentally, the problem with protocol independence is that software has not been written with the intent of being protocol-independent. Some common programs, such as Sendmail, support multiple protocols, but are still only capable of operating with certain protocols that they know about (and usually only one protocol really works). A protocol-independent application hides away knowledge of particular protocols into run-time abstractions that allow the same operations to apply regardless of what actual protocols happen to be in use. In some cases, it is not possible to make operations completely generic, in which case protocol-dependent code needs to be carefully guarded and some reasonable default actions must be available for other protocols. But most importantly, programs must be tested with several different protocols to prove that they can handle them. The jump from supporting one protocol to two is the biggest hurdle, and clearing that in a reasonable way makes supporting other protocols much easier.

2.1 Hard Coding

The most obvious protocol dependence problem seen in network programs is to hard-code the use of one protocol. Figure 1 shows an example from 4.4BSD-Lite2[3]'s telnet program. There are three major problems here. First, the protocol family to be used is hard-coded as `AF_INET`. That basically prevents protocols other than IP from being used. The family needs to be chosen based on the name resolution information, as will be discussed later. Second, the socket address used is a protocol-dependent address, in this case `sockaddr_in`. This structure is not big enough to hold addresses for some protocols, and in any case manipulating the fields in the structure itself is a protocol-dependent activity. Sockaddrs need to be treated as an opaque buffer manipulated by protocol-independent library functions or carefully guarded code. Third, IP-specific socket options are being used without any guards. That is, if the first two problems were fixed, the IP-specific `setsockopt` calls would still be done and they should always fail. Depending on the particular option being set, the socket option call needs to be replaced with an abstract equivalent or needs to be surrounded by a guard that skips the call if the protocol in use is not IP.

This particular bit of code also carries a common bug: it tries to be slightly protocol-independent and ends up worse off for the effort. It uses the protocol family returned by `gethostbyname()` and copies

addresses in a variable-length way, but copies that into a field within a `sockaddr_in` and later tries to `connect()` to that address using an `AF_INET` socket while specifying the length of the `sockaddr_in` as the length of the address information. If the family was something other than `AF_INET`, the `sockaddr_in` would probably not be filled in with something meaningful, and `connect()` call would probably fail regardless because the protocol family of the target address was not the same as that of the socket. As long as the only addresses that ever get returned by `gethostbyname()` are IP addresses, this practice will actually work. If addresses other than IP addresses were returned, programs written this way would break. This creates an interesting problem: interfaces that might be made protocol independent cannot be, because legacy programs don't use them correctly and changing what they return would break software. Using a new interface designed for protocol independence (like `getaddrinfo()`) and using it correctly will solve this problem.

A variation of this problem is hard coding addressing information, such as addresses and ports. Figure 2 shows an example from Sendmail 8.7.6[4]. There are three major problems here. First, the code always treats the address as a `sockaddr_in` without any guards. As in the example above, this is bad for protocol independence. Second, the code hard-codes an address and a port. While this is sometimes useful, it is usually bad practice and always bad practice when not combined with a test to check the protocol family. Third, the code explicitly specifies TCP as the transport protocol being used. This hard-codes a transport protocol and implies that only a small number of network protocols are usable (those that TCP has been made to run over). The second and third problems can be solved by using protocol-independent name resolution functions correctly.

2.2 Inflexible Storage

Another class of problems comes about when storing information needed by various protocols. This was already mentioned in the discussion of Figure 1, where not only does the use of `sockaddr_in` hard-code the address format of a particular protocol, but it also does not provide enough space to store the addresses of many protocols. The most common place where this problem comes into play is when used with `getpeername()`. Figure 3 shows an example of this from the 4.4BSD-Lite2's `fingerd` source; similar code sequences can be found in almost any server program. This code example also shows the assumption that the returned socket will be an IP socket; while originally a fair assumption, this needs to be fixed in order to be protocol-independent.

```

temp = inet_addr(hostp);
if (temp != (unsigned long) -1) {
    sin.sin_addr.s_addr = temp;
    sin.sin_family = AF_INET;
    (void) strcpy(_hostname, hostp);
    hostname = _hostname;
} else {
    host = gethostbyname(hostp);
    if (host) {
        sin.sin_family = host->h_addrtype;
#if defined(h_addr) /* In 4.3, this is a #define */
        memmove((caddr_t)&sin.sin_addr,
                host->h_addr_list[0], host->h_length);
...

net = socket(AF_INET, SOCK_STREAM, 0);
setuid(getuid());
if (net < 0) {
    perror("telnet: socket");
    return 0;
}
#if defined(IP_OPTIONS) && defined(IPPROTO_IP)
if (srp && setsockopt(net, IPPROTO_IP, IP_OPTIONS, (char *)srp, srln) <
    0)
    perror("setsockopt (IP_OPTIONS)");
#endif
...

if (connect(net, (struct sockaddr *)&sin, sizeof (sin)) < 0) {

```

Figure 1: Hard-Coding the Network Protocol (4.4BSD Telnet)

```

if (DaemonAddr.sin.sin_family == 0)
    DaemonAddr.sin.sin_family = AF_INET;
if (DaemonAddr.sin.sin_addr.s_addr == 0)
    DaemonAddr.sin.sin_addr.s_addr = INADDR_ANY;
if (DaemonAddr.sin.sin_port == 0)
{
    register struct servent *sp;

    sp = getservbyname("smtp", "tcp");
    if (sp == NULL)
    {
        syserr("554 service \"smtp\" unknown");
        DaemonAddr.sin.sin_port = htons(25);
    }
    else
        DaemonAddr.sin.sin_port = sp->s_port;
}

```

Figure 2: Hard Coding Addresses and Ports (Sendmail 8.7.6)

```

struct sockaddr_in sin;
...
if (logging) {
    sval = sizeof(sin);
    if (getpeername(0, (struct sockaddr *)&sin, &sval) < 0)
        err("getpeername: %s", strerror(errno));
    if (hp = gethostbyaddr((char *)&sin.sin_addr.s_addr,
        sizeof(sin.sin_addr.s_addr), AF_INET))
        lp = hp->h_name;
    else
        lp = inet_ntoa(sin.sin_addr);
    syslog(LOG_NOTICE, "query from %s", lp);
}

```

Figure 3: Use of a `sockaddr_in` to Store Arbitrary Addresses (4.4BSD fingerd)

A similar problem is also seen frequently in servers: the use of a generic `sockaddr` to store address information. Like the IP protocol specific structure, it is not big enough to hold addresses for many protocols (on most systems, the two structures are actually the same size). When the size of the address to be stored is known, a buffer of that size can be allocated. When it is not, a maximal-length buffer can be allocated using a `sockaddr_storage`, which will be discussed later.

A particularly bad special case of this problem comes about in some IP-only programs. Because IP addresses happen to be 32 bit unsigned integers and many modern systems have that as a native data type, some programs simply use integers to store IP addresses. Figure 4 shows an example from `vat 4.0b2[5]`, which uses `u_int32_t`s internally to store network addresses (this is a bit less bad than using more generic integer types, but still hopelessly IP- dependent). Due to a particularly common example of this in earlier versions of BSD, this is sometimes referred to as the “all the world’s a `u_long`” problem, and has a lot in common with the old “all the world’s a VAX” problem. Optimizing assumptions are being made about the size and form of an address that happen to work on most currently interesting systems and protocols. But they’re still poor assumptions that break portability, both in terms of supporting different systems and supporting different protocols. 4.4BSD-Lite2 has fixed this problem in many places by using `in_addr` instead, which is still protocol-dependent but at least is the correct type. In general, raw addresses should not be stored – socket addresses should be used instead.

Also, some protocols have variable-length addresses. Most existing programs treat addresses as fixed-length objects and do not store the real length as provided by run-time functions. Programs must store the length of addresses along with the addresses themselves – as with the address type, this can be necessary information for interpreting the address. This also means that

the sizes of buffers used to hold addresses should not be arbitrarily bounded.

Using the generic `sockaddr` or the wrong protocol-specific structure also creates problems with alignment. Most network protocols have some alignment requirement for their protocol-specific address structures that may not be satisfied by other structures. Care must be taken to either use the correct protocol specific address structure or to arrange for the buffer used to store addresses to be properly aligned.

The generic `sockaddr` *should* be used as a structure to which an arbitrary socket address can be cast in order to access the `sa_family` and `sa_len` fields. While those fields should have the same type no matter what protocol specific structure is used to access the buffer, it is still good use of types to use the generic `sockaddr` for access where the network protocol in use are not yet known, rather than to using the wrong protocol-specific type.

Finally, many programs assume that a “port” is an integer. The concept of an integer port number is not universal. Some protocols use string service names instead, or use other formats that are at least convertible to a string. Service endpoints should be represented as strings that may or may not end up converted to another format for representation in a socket address.

2.3 Inflexible User Interface

Many programs get address information through some sort of user interface or user parameter syntax. For example, web clients get resource information through URLs, free network programs such as `tcp_wrappers` [6] read configuration files, and some GUI network programs use four three-digit entry blanks to enter a numeric address. In many cases, the user interface or syntax that these clients use is dependent on the syntax of particular kinds of addresses. The use of colons and

slashes in URLs, for example, makes it difficult to use those characters in network addresses (colons are the delimiter for IPv6 addresses, so this is a real problem). Similarly, the configuration files for `tcp-wrappers` 7.6 uses colons as the field separator. The generic solution to this problem is to provide support for escaping and/or quoting so that somewhat arbitrary characters can be used in address information. In many cases, a quick-fix solution can be made by changing the fields or delimiters, but this is not backwards compatible and only fixes things until the next new address syntax comes along.

Another problem is that more information may need to be provided to a program in order for it to know how to correctly interpret an address. For example, a host name may be valid in multiple protocol families. If the user wants to use a specific protocol, then this needs to be somehow specified. Programs need to have enough flexibility in their user interfaces to add these sort of options.

2.4 Not Handling Multiple Addresses

Multi-homed IP support in programs is still not as common as it should be. The problem of multi-homing support – supporting multiple interfaces with one or more IP address – is similar to the problem of supporting multiple addresses for different network protocols. In both cases, some subset of the available addresses may need to be listened on (for a server) or be available for an outgoing connection. The selection of these addresses creates interesting problems. In particular, it sometimes (esp. in the case of servers) creates the need to have multiple network sockets open at once and to handle traffic on any of them. Most networking programs are written to use only one network socket, and changing that requires significant work.

Both multi-homing and multi-protocol support requires extra user interface capability. For example, both will cause multiple addresses to be returned from some name lookups, of which a subset might be reachable endpoints. The user should be either be given a choice among this set or some attempt should be made to progress in a reasonable way (for example, trying each in sequence until one succeeds). Whatever actually happens, the user should be made aware of what's happening.

2.5 Protocols Carrying Address Information

Some IP application protocols, such as FTP and talk, pass address information over the network. This means that the application protocol will need to be modified to be multi-protocol, which generally means

adding flexibility similar to what has to be done inside a network program. Exactly how to do this is outside the scope of this document, but an example is the approach used for FTP[7]. In general, the best solution to this problem is to change the application protocol to not send address information over the network, because this practice causes problems for many network/protocol translation devices.

3 New Interfaces

The IETF's IPng working group and the POSIX p1003.1g working group have made a good bit of progress in identifying and standardizing new APIs needed to develop protocol-independent programs. Beyond this, the NRL IPv6 implementation found several other interfaces that we felt needed to be present in order to develop fully functional protocol-independent applications. In many cases, the BSD sockets interfaces were almost good enough but in practice misused. The new interfaces extend many BSD interfaces and supersede others.

The new interfaces break down into roughly two categories. The first are functions to perform name resolution operations (name to addresses, address to name) in a clean way. The second are operations to help use socket addresses (`sockaddr`s) in a clean way.

3.1 Name Resolution

Figure 5 shows a brief summary of the new name resolution interfaces.

The `getaddrinfo()` and `getnameinfo()` functions provide a protocol-independent way of mapping names to addresses information and of mapping address information back to names. Given a host name, a service name, and other information to constrain the lookup, `getaddrinfo()` returns either an integer error or a list of filled in `addrinfo` structures. Each contains the information that needs to be passed to `socket()` to open a socket as well as the information that needs to be passed to `connect()` or `sendmsg()` to reach the named endpoint.

Many programs can simply take the returned list and iterate through it, executing `socket()` and `connect()` calls with the information in each list element, until one attempt succeeds completely or the list has been exhausted. Figure 6 gives an example of how to do this. Notice how the program never needs to manipulate addresses directly. The program only needs to take information out of the `addrinfo` structure and feed it into other functions. This simple block of code is capable of obtaining a connected socket with any stream protocol that is supported in both the kernel and in the runtime library. If the runtime library

```

int Network::dorecv(u_char* buf, int len, u_int32_t& from, int fd)
{
    sockaddr_in sfrom;
    int fromlen = sizeof(sfrom);
    int cc = ::recvfrom(fd, (char*)buf, len, 0,
                       (sockaddr*)&sfrom, &fromlen);
    if (cc < 0) {
        if (errno != EWOULDBLOCK)
            perror("recvfrom");
        return (-1);
    }
    from = sfrom.sin_addr.s_addr;
}

```

Figure 4: Using an Integer for an Address (from vat 4.0b2)

```

#define AI_PASSIVE      /* Socket address is intended for bind() */
#define AI_CANONNAME    /* Request for canonical name */
#define AI_NUMERICHOST  /* Don't ever try nameservice */

struct addrinfo {
    int ai_flags;          /* input flags */
    int ai_family;        /* protocol family for socket */
    int ai_socktype;      /* socket type */
    int ai_protocol;      /* protocol for socket */
    int ai_addrlen;       /* length of socket-address */
    struct sockaddr *ai_addr; /* socket-address for socket */
    char *ai_canonname;    /* canonical name for service location (iff req) */
    struct addrinfo *ai_next; /* pointer to next in list */
};

int getaddrinfo(const char *name, const char *service,
               const struct addrinfo *req, struct addrinfo **pai);

void freeaddrinfo(struct addrinfo *ai);

char *gai_strerror(int ecode);

#define NI_MAXHOST      /* Maximum host name buffer length needed */
#define NI_MAXSERV      /* Max. service name buffer length needed */
#define NI_NUMERICHOST  /* Don't do name resolution for the host */
#define NI_NUMERICSERV  /* Don't do name resolution for the service */
#define NI_NOFQDN       /* Don't fully qualify host names */
#define NI_NAMEREQD     /* Fail if name resolution for the host fails */
#define NI_DGRAM        /* Service is for a DGRAM socket (not a STREAM) */

int getnameinfo(const struct sockaddr *sa, size_t addrlen, char *host,
               size_t hostlen, char *serv, size_t servlen, int flags);

int nrl_afnametonom(const char *name); /* (Nonstandard) */
const char *nrl_afnumtoname(int num); /* (Nonstandard) */
int nrl_socktypenametonom(const char *name); /* (Nonstandard) */
const char *nrl_socktypenumtoname(int num); /* (Nonstandard) */

```

Figure 5: Summary of New Name Resolution APIs

does not support a protocol, it will not be returned by `getaddrinfo()`. If the kernel does not support a protocol, this function will print an error for those sockets and skip that protocol. This is especially important for binaries to be shipped on systems where the protocols available in the runtime library and/or kernel can be configured by the end user; one binary will be able to work as long as the system is configured so that there is one protocol that the entire system supports.

Note that `getaddrinfo()` and `getnameinfo()` handle both host names and printable numeric addresses, as appropriate. One historical problem with functions like `gethostbyname()` and `gethostbyaddr()` is that on some systems they handle printable numeric addresses and on some systems they do not. Portable programs must be written to attempt printable-numeric conversion separately, just in case – programs that assume the system handles these have encountered portability problems. Some programs have bugs caused by the old printable numeric conversion functions, making this even more of a problem. These new functions should hopefully put these problem to rest.

As shown in the example, the `gai_strerror()` function converts the errors returned by `getaddrinfo()` and `getnameinfo()` into human-printable form. There are also constants for the error values, but few programs need to distinguish between the types of failures beyond giving an appropriate error message. The `freeaddrinfo()` function releases the memory used by the result list, and *must* be called when the result is no longer needed.

The functions `nrl_afnametonum()` and `nrl_afnumtoname()` convert address family names (inet, inet6, local, etc.) to numbers and back. This is needed in order to support user entry of an address family to constrain `getaddrinfo()` lookups. For example, many NRL IPv6-enabled applications support a command line flag that the user can use to specify a family, such as “inet” or “inet6,” that selects what protocol to use. The number-to-name function is also helpful for diagnostic output. Similarly, the functions `nrl_socktypenametounum()` and `nrl_socktypenumtoname()` convert socket type names (stream, dgram, seqpacket, etc.) to numbers and back. These are less useful for user input, but are still useful for diagnostic purposes.

3.2 Socket Addresses

Figure 7 shows a brief summary of the new socket address interfaces. The major new addition is the `sockaddr_storage`, which is defined as a structure that is big enough to hold any socket address that the system supports or might support in the future, and provides sufficient alignment for any socket address that the system supports or might support in

the future. In practice, the size of the structure is bounded on many systems by the capacity of the eight bit integer used in the `sa_len` field of all socket addresses. On other systems, the bound might be provided by other structures’ fields. The bound actually chosen can be selected by the systems’ authors, but the `sockaddr_storage` is defined to have whatever size is needed. The alignment provided by the `sockaddr_storage` will typically be the largest alignment available on the system, though again the exact choice is up to the systems’ implementors.

Note that the `sockaddr_storage` is required to have fields of the same type and in the same place as the `sa_len` and `sa_family` fields in the systems’ `sockaddrs`, but that the standards that specify this data structure don not actually require those fields to have a known name and give examples with names that makes them “hidden.” While it is hoped that the long term solution will be to fix this problem in the standards, the short term most portable way to use these fields is to cast a `sockaddr_storage` to a `sockaddr` and to use the fields through the latter type.

The `sockaddr_storage` is used where a socket address needs to be stored before its exact length is known. Figure 8 shows some of the example in Figure 3, changed to take advantage of this structure as well as `getnameinfo()`. The code is not very different, but the use of the `sockaddr_storage` guarantees that any protocol-specific socket address can be safely stored in the buffer.

A controversial API extension that was used heavily in the NRL code is the `SA_LEN()` macro. On systems whose `sockaddr` has a `sa_len` field, this expands to return the contents of that field and has the same semantics except that it is only defined to be an rvalue. On systems whose `sockaddr` does not, this expands into an operation that returns the correct value based on the value of the `sa_family` field. This macro solves the problem of needing a `sockaddr`’s length for many function calls well after existing code has lost the length information. It is frequently far easier to replace a hard coded value such as `sizeof(struct sockaddr_in)` with a macro use like `SA_LEN(sa)` than it is to gut an entire program and fix this. Using the macro, this technique is portable to systems with and without `sa_len` support. Authors who have used this technique extensively have been quite supportive of it, while authors of systems that don’t have `sa_len` fields have been opposed to it.

3.3 State of Deployment

New API functions are fine as long as they are available everywhere, but deployment does not happen quickly. The interfaces described as non-standard are just that, and are unlikely to be present any-

```

int get_stream(char *host, *service)
{
    int error, fd;
    struct addrinfo req, *ai, *ai2;
    char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

    memset(&req, 0, sizeof(struct addrinfo));
    req.ai_socktype = SOCK_STREAM;

    if (error = getaddrinfo(host, service, NULL, &ai)) {
        fprintf(stderr, "getaddrinfo(%s, %s, ...): %s(%d)", gai_strerror(error),
            error);
        return -1;
    }

    for (ai2 = ai; ai = ai->ai_next) {
        if (error = getnameinfo(ai->ai_addr, ai->ai_addrlen, hbuf, sizeof(hbuf),
            sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV)) {
            fprintf(stderr, "getnameinfo(%p, %d, %p, %d, %p, %d, %d): %s(%d)\n",
                ai->ai_addr, ai->ai_addrlen, hbuf, sizeof(hbuf), sbuf, sizeof(sbuf),
                NI_NUMERICHOST | NI_NUMERICSERV, gai_strerror(error), error);
            continue;
        }

        fprintf(stdout, "Trying %s.%s...\n", hbuf, sbuf);

        if ((fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol)) < 0) {
            fprintf(stderr, "socket(%d, %d, %d): %s(%d)\n", ai->ai_family,
                ai->ai_socktype, ai->ai_protocol, strerror(errno), errno);
            continue;
        }

        if (connect(fd, ai->ai_addr, ai->ai_addrlen) < 0) {
            fprintf(stderr, "connect(%d, %p, %d): %s(%d)\n", fd, ai->ai_addr,
                ai->ai_addrlen, strerror(errno), errno);
            close(fd);
            continue;
        }

        freeaddrinfo(ai2);
        return fd;
    }

    freeaddrinfo(ai2);
    fprintf(stderr, "No connections result.\n");
    return -1;
}

```

Figure 6: Using getaddrinfo() to Get one Stream Connection

```

struct sockaddr_storage { /* Slightly nonstandard - See text for a warning */
    u_int8_t    ss_len;      /* address length */
    sa_family_t ss_family;    /* address family */
    /* other fields guarantee size and padding */
};

#define SA_LEN(sa) ((sa)->sa_len) /* Nonstandard */

```

Figure 7: New Socket Address Interfaces

```

struct sockaddr_storage ss;

...

if (logging) {
    sval = sizeof(sockaddr_storage);
    if (getpeername(0, (struct sockaddr *)&ss, &sval) < 0)
        err("getpeername: %s", strerror(errno));
}

```

Figure 8: Use of a sockaddr_storage to Store Arbitrary Addresses

where but systems with the NRL IPv6 code or with the (NRL-derived) Linux inet6-apps kit. The standard new interfaces are supposed to be present now in AIX, BSD/OS, FreeBSD, Linux (with GNU libc 2.1), OpenBSD, NetBSD, Solaris, and Tru64 UNIX. In addition, IRIX and HP-UX will probably adopt these functions very quickly (if they haven't already). In summary, all modern UNIX systems are expected to support the standardized interfaces now or very soon.

But what about legacy systems? The good news is that there are fairly portable implementations of these functions that run on legacy systems and can be included with programs. Several free implementations of the new interfaces exist, and are reasonably portable. There are already some free software packages, such as Zmailer, that have used this approach.

4 Conclusions

The existing BSD sockets API provides most of what is needed to write protocol-independent applications, but there are some things that needed to be added. More important is that the mechanisms for being abstract and generic be used correctly. It is not difficult to do, and the effort pays for itself in flexibility and future-proofing. It is hoped that authors of network programs will take the time to learn about protocol independence and work to use the technique in their programs.

5 Acknowledgements

The NRL IPv6 code, and our exploration of protocol independence, was a team effort. Other than myself, the implementation team includes or has in-

cluded Randall Atkinson, Ken Chin, Daniel McDonald, Ronald Lee, Bao Phan, Chris Telfer, and Chris Winters.

When I first pushed people to look at protocol independence issues in IPv6 API discussions, W. Richard Stevens, Jun-Ichiro "itojun" Hagino, and the KAME project team provided strong support for the idea, and their support was critical to the development of these APIs.

David Greenman, Kevin Skadron, and Chris Telfer reviewed early drafts of this paper and provided valuable feedback.

References

- [1] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6, RFC 2553, March 1999.
- [2] Randall J. Atkinson, Ken E. Chin, Bao G. Phan, Daniel L. McDonald, and Craig Metz. Implementation of IPv6 in 4.4BSD. *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.
- [3] Regents of the University of California. 4.4bsd-lite2 software distribution, 1995.
- [4] Eric Allman et al. Sendmail, version 8.7.6, 1999.
- [5] Van Jacobsen et al. Vat, version 4.0b2, 1996.
- [6] Vietse Venema et al. tcp_wrappers, version 7.6, 1997.
- [7] M. Allman, S. Ostermann, and C. Metz. FTP Extensions for IPv6 and NATs, rfc 2428, September 1998.

Scalable Network I/O in Linux

Niels Provos, *University of Michigan*
provos@citi.umich.edu

Chuck Lever, *Sun-Netscape Alliance*
chuckl@netscape.com

Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu
<http://www.citi.umich.edu/projects/linux-scalability>

Abstract

Recent highly publicized benchmarks have suggested that Linux systems do not scale as well as other systems, such as Windows NT, when used as network servers. Windows NT contains features such as I/O completion ports that help boost network server performance and scalability. In this paper we focus on improving the Linux implementation of `poll()` to reduce the expense of managing large numbers of network connections. We also explore the newer POSIX RT signal API that will help network servers scale into the next decade. A comparison between the two interfaces shows that a server using our `/dev/poll` interface scales better than a server using RT signals.

1. Introduction

Many traditional web server benchmarks have focused on improving throughput for clients attached to the server via a high-speed local area network [13]. Recent studies have shown, however, that the difference between 32 high performance clients connected via gigabit Ethernet, and 32,000 high latency, low bandwidth connections from across the Internet, is extremely important to server scalability [8]. Connections that last only seconds do not place the same load on a server that slow error-prone connections do, due to resources consumed by error recovery, and the expense of managing many connections at once, most of which are idle.

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

This document is Copyright © 2000 by AOL-Netscape, Inc., and by the Regents of the University of Michigan. Trademarked material referenced in this document is copyright by its respective owner.

Experts on server architecture have argued that servers making use of a combination of asynchronous events and `poll()` are significantly more scalable than today's servers in these environments [2, 3, 5]. In Linux, signals can deliver I/O-completion events. Unlike traditional UNIX signals, POSIX Real-Time (RT) signals carry a data payload, such as a specific file descriptor that recently changed state. Signals with a payload enable network server applications to respond immediately to network requests, as if they were event driven. An added benefit of RT signals is that they can be queued in the kernel and delivered to an application one at a time, in order, leaving an application free to collect and process events when convenient.

The RT signal queue is a limited resource. When it is exhausted, the kernel signals a server to switch to polling, which delivers multiple completion events at once. Normally in a server like this, polling is simply an error recovery mechanism. However, the size of the RT signal queue might also be used as a load threshold to help

network servers determine whether RT signals or the `poll()` interface is more efficient.

We have identified two areas of study. First, we demonstrate several modifications that improve `poll()`'s scalability and performance when a large proportion of a server's connections are inactive. Second, finding the right combination of RT signals and polling might allow network servers to leverage the latency advantages of completion notification against the throughput boosts of using `poll()`.

In this paper, we outline several improvements to the `poll()` interface, and measure the performance change of application using the improved `poll()`. We describe a test harness that simulates loads consisting of many inactive connections. We use this test harness to measure changes in application throughput as we vary the server's event model. Finally we report on our experiences using the new POSIX RT signals.

2. Using POSIX RT Signals: Introducing `phhttpd`

`Phhttpd` is a static-content caching front end for full-service web servers such as Apache [2, 10]. Brown created `phhttpd` to demonstrate the POSIX RT signal mechanism added to the Linux kernel starting in the late 2.1.x kernel development series, and completed during the 2.3.x series. POSIX RT signals provide an event delivery system by allowing an application to assign unique signal numbers to each open file descriptor using `fcntl(fd, F_SETSIG, signum)`. The kernel raises the assigned signal whenever a `read()`, `write()`, or `close()` operation completes.

```
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

Figure 1. Standard `pollfd` struct

```
struct siginfo {
    int si_signo;
    int si_errno;
    int si_code;
    union {
        /* other members elided */
        struct {
            int _band;
            int _fd;
        } _sigpoll;
    } _sifields;
} siginfo_t;
```

Figure 2. Simplified `siginfo` struct.

To avoid complexity and race conditions, the chosen RT signals are masked during normal server operation.

An application uses `sigwaitinfo()` to pick up pending signals from the RT signal queue. `Sigwaitinfo()` returns a `siginfo` struct (see FIG. 2) for a single event. The `_fd` and `_band` fields in this struct contain the same information as the `fd` and `revents` fields in a `pollfd` struct (see FIG. 1).

The kernel raises `SIGIO` if the RT signal queue overflows. An application then flushes pending signals by changing their signal handler to `SIG_DFL`, and to recover, it uses `poll()` to discover any remaining pending activity.

Events queued before an application closes a connection will remain on the RT signal queue, and must be processed and/or ignored by applications. For instance, when a socket closes, a server application may receive and try to process previously queued read or write events before it picks up the close event, causing it to attempt inappropriate operations on the closed socket.

3. `poll()` Optimizations

There are two motivations for improving `poll()`. First, many legacy applications can benefit, with little or no modification, from performance and scalability improvements in `poll()`. While the changes necessary to take advantage of a new interface might be few, an overall architectural update for legacy applications is usually unnecessary. Yet updating an application to use POSIX RT signals is a major overhaul with a concomitant increase in complexity. A second purpose for improving `poll()` is that even with POSIX RT signals, `poll()` is still required to handle special cases such as RT signal queue overflows. An efficient `poll()` implementation helps performance and scalability of the new paradigm.

For the `poll()` interface to maintain performance comparable to the newer POSIX RT signal interfaces, it needs a face lift. We've enhanced `poll()` by making the following optimizations:

- We provide an interface that maintains state in the kernel, so state doesn't have to be passed in during every `poll()` invocation
- We allow device drivers to post completion events to `poll()`, reducing the need to invoke device-specific poll operations when scanning for events
- We eliminate result copying when `poll()` returns by creating a special address space mapping that is shared between the kernel and the application

In this section we describe these changes and evaluate their respective performance implications.

3.1 Maintaining State in the Kernel

To invoke `poll()` an application must build a set of interests, where an interest is a file descriptor that may have I/O ready, then notify the kernel of all interests by passing it the complete set of interests via `poll()`. As the number of interests increases, this mechanism becomes unwieldy and inefficient. The entire set must be copied into the kernel upon system call entry. The kernel must parse the entire interest set to carry out the request. Then each interest must be checked individually to assess its readiness.

Banga, Druschel, and Mogul have described new operating system features to mitigate these overheads [4]. They suggest that the `poll()` interface itself can be broken into one interface used to incrementally build an application's interest set within the kernel, and another interface used to wait for the next event. They refer to the first interface as `declare_interest()`, while the second is much like today's `poll()`. Using `declare_interest()`, an application can build its interest set inside the kernel as connections are set up and torn down. The complete interest set is never copied between user space and kernel space, completely eliminating unnecessary data copies (for instance, when there is no change in the interest set between two `poll()` invocations).

Recent versions of Solaris include a similar interface called `/dev/poll` [6]. This character device allows an application to notify the kernel of event interests and to build a (potentially) very large set of interests while reducing data copying between user space and kernel space. As far as we know this is the first real implementation of `declare_interest()`. We chose to implement this because, if it is effective, it will allow easier portability of high-performance network applications between Solaris and Linux.

An application opens `/dev/poll` and receives a file descriptor. The kernel associates an interest set with this file descriptor. A process may open `/dev/poll` more than once to build multiple independent interest sets. An application uses `write()` operations on `/dev/poll` to maintain each interest set.

Writing to `/dev/poll` allows an application to add, modify, and remove interests from an interest set. Applications construct an array of standard `pollfd` structs, one for each file descriptor in which it is interested (see Figure 1). Enabling the `POLLREMOVE` flag in the `events` field indicates the removal of an interest. Specifying a file descriptor the kernel already knows about allows an application to modify its interest. The contents of the `events` field replace the previous interest, unlike the Solaris implementation, where the

`events` field is OR'd with the current interest. If complete Solaris compatibility is desired, this behavior can be adjusted with a minor modification to the device driver.

```
struct dvpoll {
    struct pollfd* dp_fds;
    int dp_nfds;
    int dp_timeout;
}
```

Figure 3. `dvpoll` struct

To wait for I/O events, an application issues an `ioctl()` with a `dvpoll` struct (see FIG. 3). This struct indicates how long to wait, and specifies a return area for the results of the poll operation. In general, only a small subset of an application's interest set becomes ready for I/O during a given poll request, so this interface tends to scale well.

A hash table contains each interest set within the kernel. On average, hash tables provide fast lookup, insertion, and deletion. For simplicity, when the average bucket size is two, the number of buckets in the hash table is doubled. The hash table is never shrunk.

3.2 Device Driver Hints

When an application registers interest in events on file descriptors with the `poll()` system call, the kernel passes this information to device drivers and puts the process to sleep until a relevant event occurs. When an application process wakes up, the kernel must scan all file descriptors in which the application has registered interest to check for status changes. This is the case even though the status of only one file descriptor in hundreds or thousands might have changed.

Now that we have an efficient mechanism for applications to indicate their interests, it would be useful if device drivers could indicate efficiently which file descriptors changed their status. We extend the `/dev/poll` implementation to make file descriptor information available to device drivers. The `/dev/poll` implementation maintains this information in a backmapping list. When an event occurs, the driver marks the appropriate file descriptor for each process in its backmapping list. When `poll()` scans an interest set to pick up new work, it uses this hint to avoid an expensive call to the device driver's poll callback. When managing a large number of high latency connections, this greatly reduces the number of driver poll operations that show that nothing has changed.

Hints allow `poll()` to determine if a cached result from a previous poll call is still valid. Specifically, a hint indicates a change in the socket's status, so it is time to invoke the device driver's poll callback. This also erases the current hint. We cache the result re-

turned by the device driver, in the hope that we can reuse it without having to invoke the poll callback again soon. We do not receive hints that indicate the change from ready to not-ready, however. This means that a cached result indicating readiness has to be reevaluated each time.

To prevent the hinting system from requiring every device driver to be modified, device drivers indicate whether they support hinting. In this way, only essential drivers must be modified, *e.g.* network device drivers.

At this point, all backmapping lists are protected by a single read-write lock. Hints require only a read lock, so the lock itself is generally not contended. The lock is held for writing only when the interest set is modified. Each socket gets its own backmap, so ideally the backmap lock should be added in a per-socket structure to reduce lock contention and improve cache line sharing characteristics on SMP hardware. Each per-socket lock requires an extra 8 bytes.

3.3 Reducing Result Copy Operations

As the list of file descriptors passed to poll() grows, the overhead to copy out and parse the results also increases. To reduce this overhead, we need to improve the way the kernel reports the results of a poll() operation. The safest and most efficient way to do this is to create a memory map shared between the application and kernel where the kernel may deposit the results of the poll() operation.

We added this feature to our /dev/poll implementation. The application invokes mmap() on /dev/poll to create the mapping. Results from poll() for that process are reported in that area until it is munmap()'d by the application. Usually, the size of the result set is small compared to the size of the entire interest set, so we do not expect this modification to make as significant an impact as /dev/poll and device driver hints.

To create the result area, an application invokes ioctl(DP_ALLOC) to allocate room for a specific number of file descriptor results. This is followed by an mmap() call on a previously opened /dev/poll file descriptor to share the mapping between the kernel and the application. When polling, an application uses ioctl(DP_POLL) and specifies a NULL in the dp_fds field (see FIG. 2). When the application is finished, it uses munmap() to deallocate the area, then it closes /dev/poll normally.

4. POSIX Real-Time Signals v. poll()

A fundamental question is how great a server workload is required before polling is a more efficient way to gather requests. Are there *any* times when polling is a

better choice? In this section we address the following questions:

- How big is the difference in performance between POSIX RT signals and poll(). Naturally this depends on the application implementation, but something as crude as an order of magnitude number would still be useful. What are the factors that determine this difference in performance—inefficiencies in poll() itself, argument copying, and so on?
- How important is an efficient poll() implementation for good overall performance of an implementation based on POSIX RT signals?
- How complete is the POSIX RT signal interface and implementation? Is it easy to use? Are there races or performance issues? Is it easy to use in combination with threads and black-box libraries?

To study these questions, we compare the performance of polling and event-driven architectures with a benchmark. The benchmark indicates which parts of the performance curve are served better by a particular event model. Imagine a hybrid server that can switch between polling and processing incoming requests via RT signals.

- To reduce the latencies of polling models, the server uses RT signals to process incoming requests and to handle them as soon as they arrive.
- To manage resource exhaustion in the kernel, the server uses RT signals until the signal queue reaches its maximum length.
- To overcome the inefficiencies of one-at-a-time event handling, the server uses polling after its workload becomes heavy.

Such a server might use the RT signal queue maximum as a crossover point for two reasons. First, it is built into the RT signal interface. When the signal queue overflows, the application receives a signal indicating that the overflow occurred. A poll() is necessary at this point to make sure that no requests are dropped. Second, the queue length tracks server workload fairly well. As server workload increases, so does the RT signal queue length. Thus it becomes an obvious indicator to cause a workload-triggered switch between event-driven and polling modes.

By studying the behavior and performance at the crossover point between RT signals and polling in a hybrid server, we gain an understanding of each design's relative advantages. Before creating such an imaginary hybrid, we can run specific tests that show whether each model has appropriate complementary performance and scalability characteristics.

Additionally, in real servers using the RT signal queue, we'd like to be sure that queue overload recovery mechanisms (*i.e.* invoking `poll()` to clean up) do not make the overload situation worse due to poor performance. Even better, perhaps `poll()` can perform well enough relative to POSIX RT signals that we don't have to relegate it to managing overloads. Note that the RT signal queue maximum length is normally set high enough (1024 by default) that it is never exceeded in today's implementations.

5. Benchmark

Our test harness consists of two machines running Linux connected via a 100 Mbit/s Ethernet switch. The workload is driven by an Intel SC450NX with four 500MHZ Xeon Pentium III processors (512Kb of L2 cache each), 512Mb of RAM, and a pair of SYMBIOS 53C896 SCSI controllers managing several LVD 10KRPM drives. Our web server runs on custom-built hardware equipped with a single 400MHZ AMD K6-2 processor, 64Mb of RAM, and a single 8G 7.2KRPM IDE drive. The server hardware is small so that we can easily drive the server into overload. We also want to eliminate any SMP effects on our server, so it has only a single CPU.

The benchmark clients are driven by `httpperf` running on the four-way Pentium III [7]. The web servers are `thttpd`, a simple single-process event-driven web server that is easy to modify, and `phhttpd`, an experimental server created to demonstrate the POSIX RT signal interface [9, 2].

The `httpperf` benchmark client provides repeatable server workloads. We vary the server implementation and try each new idea with fixed workloads. We are most interested in static content delivery as that exercises the system components we are interested in improving. A side benefit of these improvements is better dynamic content service.

Scalability is especially critical to modern network service when serving many high-latency connections. Most clients are connected to the Internet via high-latency connections, such as modems, whereas servers are usually connected to the Internet via a few high bandwidth low-latency connections. This creates resource contention on servers because connections to high-latency clients are relatively long-lived, tying up server resources, and they induce a bursty and unpredictable interrupt load on the server [8].

Most web server benchmarks don't simulate high-latency connections, which appear to cause difficult-to-handle load on real-world servers [5]. We've modified the `httpperf` benchmark to simulate these slower connections to examine the effects of our improvements on

more realistic server workloads. We add client programs that do not complete an http request. To keep the number of high-latency clients constant, these clients reopen their connection if the server times them out.

There are several system limitations that influence our benchmark procedures. There are only a limited number of file descriptors available for single processes; `httpperf` assumes that the maximum is 1024. We modified `httpperf` to cope dynamically with a large number of file descriptors. Additionally, we can have only about 60000 open sockets at a single point in time. When a socket closes it enters the `TIMEWAIT` state for sixty seconds, so we must avoid reaching the port number limitation. We therefore run each benchmark for 35,000 connections, and then wait for all sockets to leave the `TIMEWAIT` state before we continue with the next benchmark run.

Our benchmark configuration contains only a single client host and a single server host, which makes the simulated workload less realistic. However, our benchmark results are strictly for comparing relative performance among our implementations. We believe the results also give an indication of real-world server performance.

A web server's static performance depends on the size distribution of requested documents. Larger documents cause sockets and their corresponding file descriptors to remain active over a longer time period. As a result the web server and kernel have to examine a larger set of descriptors, making the amortized cost of polling on a single file descriptor larger. In our tests, we request a 6 Kbyte document, a typical `index.html` file from the CITI web site.

5.1 /dev/poll benchmark results

Our first series of benchmarks measures the scalability of using `/dev/poll` instead of the stock version of `poll()`. We use `httpperf` to drive a uniprocessor web server running `thttpd`.

We run two series of tests. First, we test stock `thttpd` running on stock Linux 2.2.14, varying the load offered by `httpperf` by adjusting the number of inactive connections. The second test is the same, but replaces the kernel with a 2.2.14 kernel that supports `/dev/poll`, and replaces `thttpd` with a version modified to use `/dev/poll` instead of `poll()`. A subset of the results of these two series of tests is shown in FIGS. 4 through 10. Each of these graphs represents data from a single run of the benchmark.

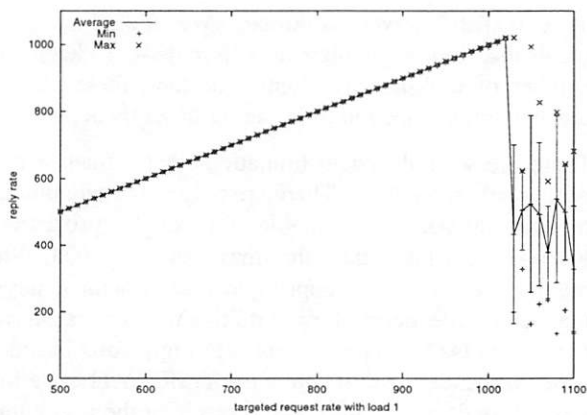


FIGURE 4. Normal `httpd` using normal `poll()`, with one extra inactive connection. As expected, the server performs well when processing only active connections. After reaching a high enough request rate however, server performance breaks down as processing latency begins to exceed request rate.

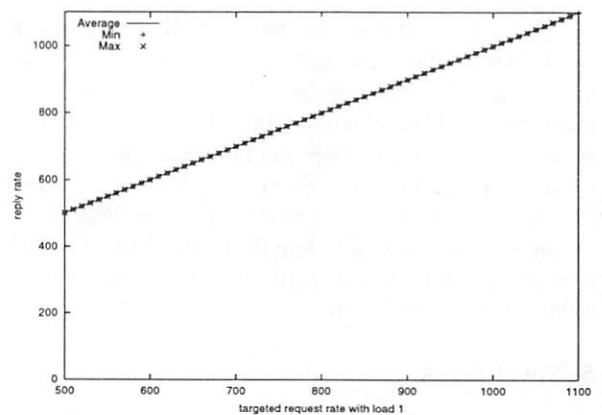


FIGURE 5. `httpd` modified to use `/dev/poll`, with one extra inactive connection. With no inactive connections, the modified server performs well at all request rates. Unlike stock `httpd`, there does not appear to be any point where processing latency exceeds request rate.

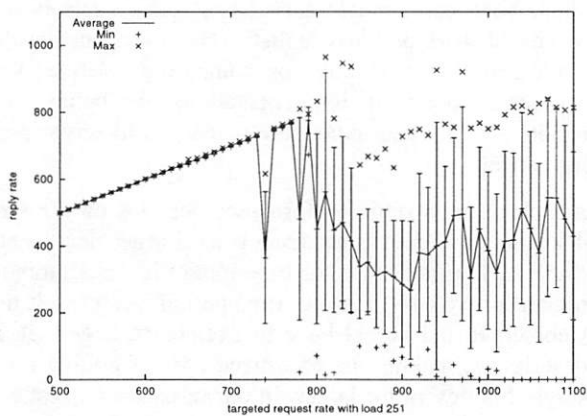


FIGURE 6. Normal `httpd` using normal `poll()`, with 251 extra inactive connections. As load caused by inactive connections increases, processing latencies likewise increase. Server performance breaks down sooner, causing minimum response rates of zero in several places.

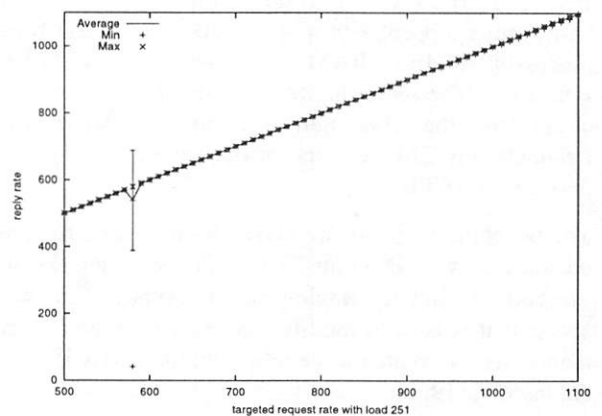


FIGURE 7. `httpd` modified to use `/dev/poll`, with 251 extra inactive connections. With some inactive connections, the modified server performs almost as well as a server with no inactive connections.

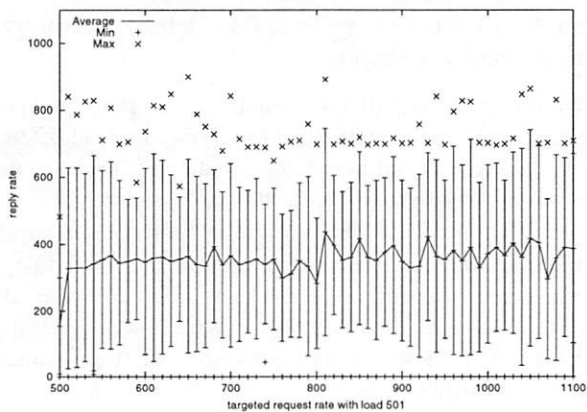


FIGURE 8. Normal `httpd` using normal `poll()`, with 501 extra inactive connections. Latency due to processing inactive connections dominates server performance for all request rates, causing poor performance and high error rates.

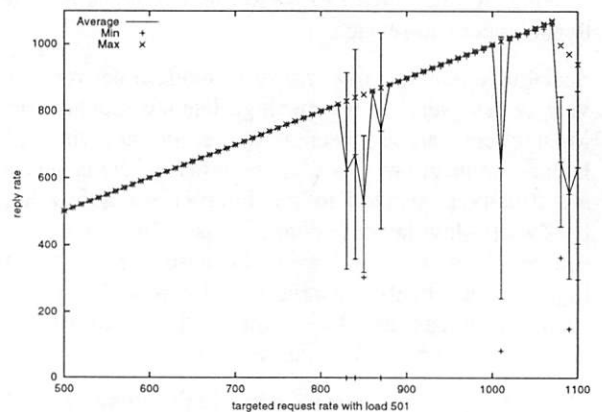


FIGURE 9. `httpd` modified to use `/dev/poll`, with 501 extra inactive connections. Despite some response rate anomalies, the modified server manages a high inactive connection load with ease. Performance begins to break down at extreme high request rates.

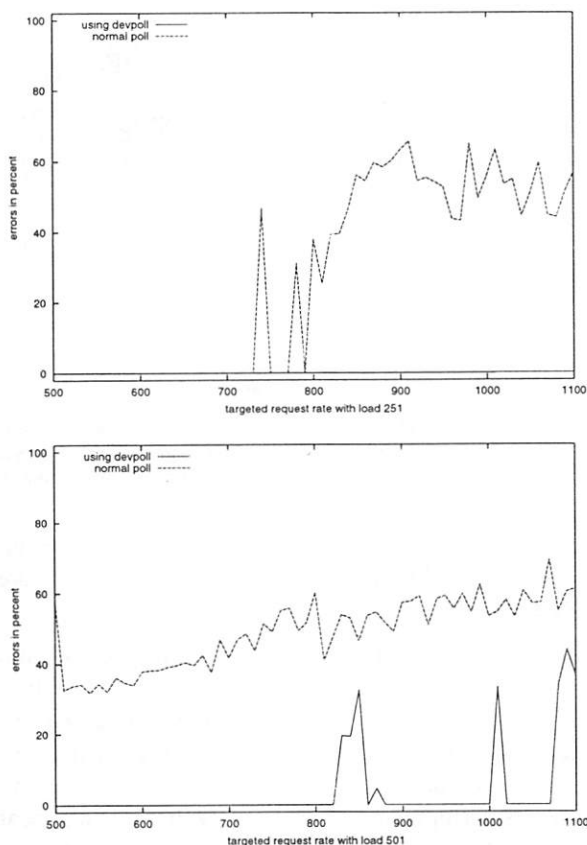


FIGURE 10. Error rate reported by `httperf` for stock `thttpd` and for `thttpd` modified to use `/dev/poll`. `httperf` maintains 251 inactive connections during the test shown in the top graph, and 501 inactive connections during the test shown in the bottom graph. `thttpd` using `/dev/poll` runs the test with 251 inactive connections with no errors whatsoever.

To simulate more realistic load on the server, we use an extra program to create inactive server connections. FIGS. 4 through 9 show the results of the benchmark for stock and modified `thttpd` with 1, 251 and 501 inactive connections. The graphs on the left show the results for stock `thttpd` using normal `poll()`. The graphs on the right the results for `thttpd` modified to use `/dev/poll`. Each graph plots the average response rate with error bars showing standard deviation against the request rate generated by the benchmark client. Ideally the generated request rate should match the server's response rate. The minimum and the maximum response rate for each run are also provided for comparison.

We observe a decrease in the average response rate as the number of inactive connections increases for both versions of `thttpd`. Some graphs show jumps in the maximum measured response rate while the minimum rate approaches zero, indicating that the server starves some connections.

`thttpd` using `/dev/poll` fully or partially achieves the desired response rate for all offered loads, as indicated by the data points showing maximum achieved response rate. On the other hand, the unmodified server is unable to maintain its throughput with increasing inactive connection load or increasing request rate. Its average response rate is smaller in all cases compared to `/dev/poll`. Banga and Drushel obtain a similar result [8].

FIG. 10 plots the percentage of connections aborted due to errors during runs with 251 and 501 inactive connections. Connection errors can result when the client runs out of file descriptors, when connections time out, or when the server refuses connections for some reason. For stock `thttpd`, the error rate increases slowly to 60% of all connections. `thttpd` using `/dev/poll` experiences only sporadic errors. In fact, when using `/dev/poll`, we measured no connection errors for benchmarks with fewer than 501 inactive connections.

Both the effective reply rate and the percent of connection errors demonstrate that `thttpd` using `/dev/poll` scales better than the unmodified version using `poll()`.

5.2 Comparing event models

Our second series of benchmarks is designed to compare the benefits of an RT signal-based event core with an event core designed around `poll()`. If they scale complementarily, it makes sense to try a hybrid server that switches between the two, triggered by server load.

FIGS. 11 through 13 illustrate the scalability of an unmodified single-threaded `phhttpd` server running on custom-built hardware equipped with a single 400MHZ AMD K6-2 processor, 64Mb of RAM, and a single 8G 7.2KRPM IDE drive. Our modified `httperf` client runs on an Intel SC450NX with four 500MHZ Xeon Pentium III processors (512Kb of L2 cache each), 512Mb of RAM, and a pair of SYMBIOS 53C896 SCSI controllers managing several LVD 10KRPM drives. Both machines are attached to a 100 Mbit/s Ethernet switch. The web server runs Linux 2.2.14 with complete support for RT signals back-ported from the 2.3 kernel series. The benchmark client runs stock Linux 2.2.14. Both machines are loaded with the Red Hat 6.1 distribution.

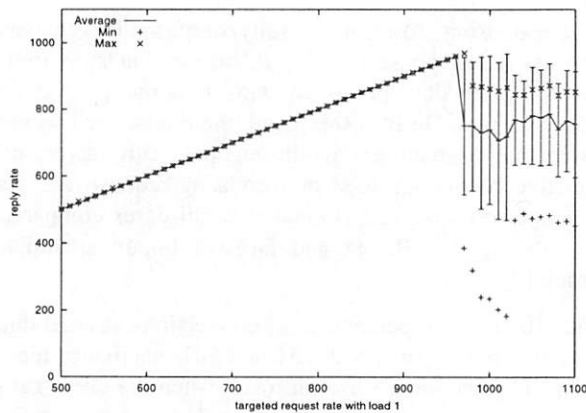


FIGURE 11. **phhttpd** with 1 extra inactive connection. Performance at lower request rates compares with the best performance of other servers. Very high request rates cause the server to falter, however. We believe this is due to the system call overhead of processing RT signals. During high loads, this overhead slows the server's ability to process requests.

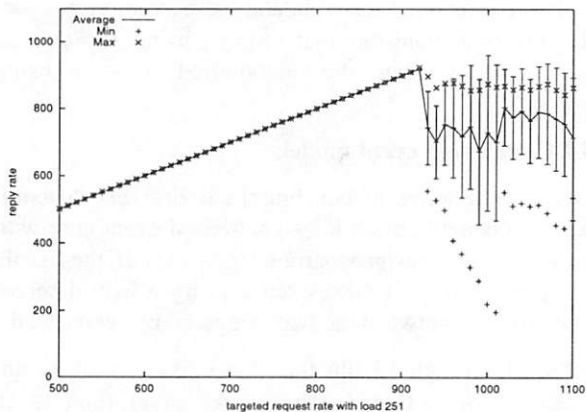


FIGURE 12. **phhttpd** with 251 extra inactive connections. With some inactive connections present, the server reaches its performance knee sooner. Inactive connections appear to increase the overhead of handling active connections, something that we didn't expect to find in a signals-based server implementation. This may be a problem with RT signals or with the **phhttpd** implementation itself.

As with the earlier `/dev/poll` benchmarks, we vary offered load by fixing the number of inactive connections, then we gradually increase the client request rate and record the corresponding server response rate. We compare a single-threaded **phhttpd** configuration against **thttpd**, a single process web server. Comparing FIGS. 11 through 13 with FIGS. 4, 6, and 8, clearly **phhttpd** outperforms the stock version of **thttpd**. However, comparing FIG. 11 to FIG. 5, we see that on the same hardware with few inactive connections, **thttpd** using `/dev/poll` responds more scalably to a higher load of active connections than does **phhttpd**.

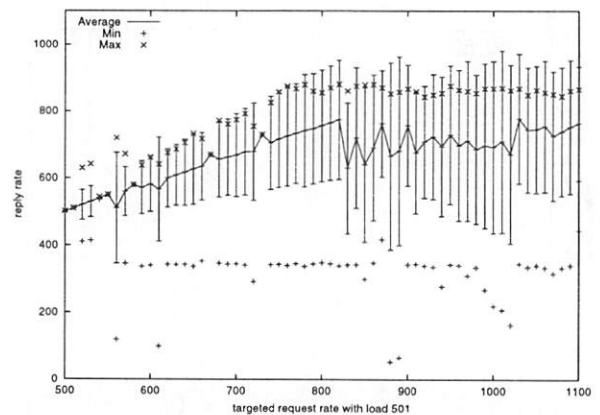


FIGURE 13. **phhttpd** with 501 extra inactive connections. In this test, load due to inactive connections appears to affect server throughput at all request rate levels. Compared to the throughput of **thttpd** using `/dev/poll`, this server scales less well.

The disparity between request and response rate increases markedly as more inactive connections are added to **phhttpd**'s load.

As FIG. 13 demonstrates, a heavy load of inactive connections causes **phhttpd** to perform worse than **thttpd** using `/dev/poll`, even at low request rates. Because **phhttpd** is unfinished and experimental, we believe that further refinements to **phhttpd** can improve its performance and scalability, but it is not clear whether it will perform better than **thttpd** based on `/dev/poll`.

An important benefit of using `/dev/poll` is that it scales well when a large number of inactive connections is present. However, even without any inactive connections `/dev/poll` scales better for high request rates compared to either stock **thttpd** or **phhttpd** using RT signals.

Another assumed advantage of RT signals is low latency. FIG. 14 shows median server response latency, in milliseconds. Median response latency evenly divides all measured responses at that load into half that are slower than the indicated result, and half that are faster. This measurement is a good reflection of a client's experience of a server's responsiveness. We see in FIG. 14 that **phhttpd** indeed serves requests with a median latency 1-3 milliseconds faster than the `/dev/poll`-based **thttpd** server across a wide range of offered load. After sufficiently high load, however, **phhttpd**'s median response latency leaps to over 120ms per request, while **thttpd**'s response increases only slightly.

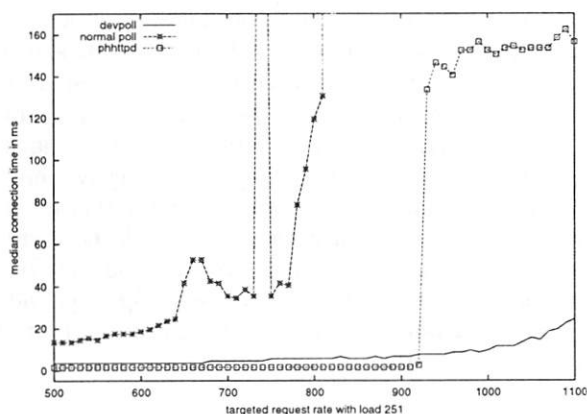


FIGURE 14. Median latency results of `phhttpd` with 251 extra inactive connections. For loads up to 900 concurrent `httperf` connections, `phhttpd` responds slightly faster than `thttpd` using `/dev/poll`. Above 900 concurrent connections, `phhttpd`'s connection latency jumps to over 120ms, whereas `thttpd`'s latency remains fairly steady. This is another indication that `thttpd` scales better than `phhttpd`.

6. Discussion and Future work

Originally we intended to modify `phhttpd` to use `/dev/poll` for these tests. After examining `phhttpd`, however, we saw that it completely rebuilds its poll interest set when recovering from RT signal queue overflow, negating any benefit to maintaining interest set state in the kernel rather than at the application level. Each thread that manages an RT signal queue for a listener socket has a partner thread that waits to handle RT signal queue overflow. When an overflow signal is raised, the thread managing the RT signal queue passes all of its current connections, including its listener socket, to its poll sibling, via a special UNIX domain socket. Considering that the server load is heavy enough to cause a queue overflow, the added work and inefficiency of transferring each connection one at a time and building a `pollfd` array from scratch will probably result in server meltdown.

When load subsides, the current `phhttpd` server does not switch from polling mode back to RT signal queue mode. Brown never implemented this logic [11].

To use either `poll()` or `/dev/poll` efficiently in `phhttpd`, we need to re-architect it. The RT signal queue overflow recovery mechanism should operate in the same thread as the RT signal queue handler. Additionally, RT signal queue processing should maintain its `pollfd` array (or corresponding kernel state) concurrently with RT signal queue activity. This would allow switching between polling and signal queue mode with very little overhead. Using `/dev/poll` without re-architecting this server won't help it scale unless it maintains its interest set concurrently with RT signal queue activity. Completely re-architecting `phhttpd` is

beyond the scope of this paper. Future work may include a reworked server based on RT signals and `/dev/poll`.

Thus, modifying applications to use the `/dev/poll` interface efficiently requires more extensive changes to legacy applications than we had hoped. Applications of this type often entirely rebuild their `pollfd` array each time they invoke `poll()`, as `phhttpd` does.

Application developers may be tempted to treat POSIX RT signals like an interrupt delivery system. When used with signal handlers, signal delivery is immediate and asynchronous. However, when they are left masked and are picked up via `sigwaitinfo()`, POSIX RT signals behave much like `poll()`. The information delivered by a `siginfo` struct is the same as that in a `pollfd` struct, and, like `poll()`, it is provided synchronously when the application asks for it.

With `poll()`, however, the amount of data stored in the kernel is always bounded, because information about current activity on a file descriptor *replaces* previous information. However, managing this data in the kernel can become complex and inefficient as an application's interest set increases in size.

The POSIX RT signal queue receives a *new* item for any connection state change in a given interest set, and this item is simply added to the end of a queue. This necessitates a maximum queue limit and a special mechanism for recovering from queue overflow. Quite a bit of time can pass between when the kernel queues an RT signal and when an application finally picks it up. Sources of latency are varied: the kernel may need to swap in a stack frame to deliver a signal, lock contention can delay an application's response, or an application may be busy filling other requests. This means that a server picking up a signal must be prepared to find the corresponding connection in a different state. Later state changes that reflect the current state of the connection may be farther down the queue.

So, like the information contained in `pollfd` structs, events generated by `sigwaitinfo()` can be treated only as hints. Several connection state changes can occur before an application gets the first queued event indicating activity on a connection. Signals dequeue in order of their assigned signal number, thus activity on lower-numbered connections can cause longer delays for activity reports on higher-numbered connections.

Another difficulty arises from the fact that the Linux threading model is incompatible with POSIX threads when it comes to catching signals. POSIX threads run together in the same process and catch the same signals, whereas Linux threads are each mapped to their own `pid`, and receive their own resources, such as signals. It

is not clear how RT signal queuing should behave in a non-Linux pthread implementation. Certainly there are some interesting portability issues here.

Several developers have observed that it is difficult to share a thread's POSIX RT signal queue among non-cooperative or black-box libraries [10, 11]. For instance, glibc's pthread implementation uses signal 32. If an application starts using pthreads after it has assigned signal 32 to a file descriptor via `fcntl()`, application behavior is undetermined. There appears to be no standard externalized function available to allocate signal numbers atomically in a non-cooperative environment.

Even when no signal queue overflow happens, the RT signal model may have an inherent inefficiency due to the number of system calls needed to handle an event on a single connection. This number may not be critical while server workload is easily handled. When the server becomes loaded, system call overhead may dominate server processing and cause enough latency that events wait a long time in the signal queue. To optimize signal handling, the kernel and the application can dequeue signals in groups instead of singly (similar to `poll()` today). We plan to implement a `sigtimedwait4()` system call which would allow the kernel to return more than one `siginfo` struct per invocation.

Future work in this area includes the addition of support in `phhttpd` for efficiently recovering from RT signal queue overflow to the signal worker thread. A closer look at `phhttpd`'s overall design may reveal weaknesses that could account for its performance in our tests. The use of specialized system calls such as `sendfile()` might also be interesting to study in combination with the new RT signal model.

There are several possible improvements to `/dev/poll`. Applications wishing to update their interest set and immediately poll on that set must use a pair of system calls, `write()` followed by `ioctl()`. A single `ioctl()` that handles both operations at once could improve efficiency. Our backmap scheme could benefit from finer grained locking, as described earlier in this paper. Sharing the result map among several threads may make a shared work queue possible. Also, improving hint caching can reduce even further the number of device driver poll operations required to obtain accurate `poll()` results.

A careful review of the current poll `wait_queue` mechanism might reveal areas for improved performance and scalability. Brown postulates that expensive `wait_queue` manipulation is where POSIX RT signals have an advantage over `poll()` [11]. The `wait_queue` mechanism is only invoked while no internal poll op-

eration returns an event that would cause the process to wake up. Once such an event is found and it is known that the process will be awakened, the `wait_queue` is not manipulated further. To avoid `wait_queue` operations, file descriptors that have events pending should be polled first. We plan to modify our hinting system so that active connections are checked first during a poll operation. Managing each interest set with more efficient data structures in the kernel could improve performance even further. It may also help to provide the option of waking only one thread, instead of all of them.

7. Conclusion

Because of the amount of work required to poll efficiently in `phhttpd`, we were unable to directly test our theories about hybrid web servers for this paper. However, it is clear that, for our benchmark, `thhttpd` using `/dev/poll` scales better than single-threaded `phhttpd` using RT signals at both low and high inactive connection loads. Once the number of inactive connections becomes large relative to the number of active connections, the difference in performance between polling and signaling exposes itself across all request rates. Latency results at lower loads favor `phhttpd`. As load increases, however, `thhttpd` using `/dev/poll` maintains stable median response time, while `phhttpd` median response time increases by more than an order of magnitude. Surprisingly, it may never be better to use RT signals over a properly architected server using `/dev/poll`.

The POSIX RT signal interface is young, and still evolving. Today's signals-based servers are complicated by extra processing that may be unnecessary once developers understand RT signals better, and when OS implementations have improved. We expect further work in this area will improve their ease of use, performance, and scalability.

Software enhancements described herein are freely available. Please contact the authors for more information.

7.1. Acknowledgements

The authors thank Peter Honeyman and Stephen Tweedie for their guidance. We also thank the reviewers for their comments. Special thanks go to Zach Brown and Dan Kegel for their insights, and to Intel Corporation for equipment loans.

8. References

- [1] G. Banga and J. C. Mogul, "Scalable Kernel Performance for Internet Servers Under Realistic

Load,” *Proceedings of the USENIX Annual Technical Conference*, June 1998.

- [2] Z. Brown, *phhttpd*, people.redhat.com/zab/phhttpd, November 1999.
- [3] Signal driven IO (*thread*), linux-kernel mailing list, November 1999.
- [4] G. Banga, P. Drushel, J. C. Mogul, “Better Operating System Features for Faster Network Servers,” *SIGMETRICS Workshop on Internet Server Performance*, June 1998.
- [5] J. C. Hu, I. Pyarali, D. C. Schmidt, “Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-Speed Networks,” *Proceedings of the 2nd IEEE Global Internet Conference*, November 1997.
- [6] Solaris 8 man pages for `poll(7d)`. docs.sun.com:80/ab2/coll.40.6/REFMAN7/@Ab2PageView/55123?Ab2Lang=C&Ab2Enc=iso-8859-1
- [7] D. Mosberger and T. Jin, “httpperf – A Tool for Measuring Web Server Performance,” *SIGMETRICS Workshop on Internet Server Performance*, June 1998.
- [8] G. Banga and P. Druschel, “Measuring the Capacity of a Web Server,” *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [9] `thttpd` - tiny/turbo/throttling web server. www.acme.com/software/thttpd
- [10] Apache Server, The Apache Software Foundation. www.apache.org
- [11] Z. Brown, personal communication, April 2000.
- [12] J. Meyers, personal communication, May 1999.
- [13] B. Weiner, “Open Benchmark: Windows NT Server 4.0 and Linux,” www.mindcraft.com/whitepapers/openbench1.html

Accept() Scalability on Linux

Stephen P Molloy, *University of Michigan*
smolloy@engin.umich.edu

Chuck Lever, *Sun-Netscape Alliance*
chuckl@netscape.com

Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu
<http://www.citi.umich.edu/projects/linux-scalability>

Abstract

This report explores the possible effects of a "thundering herd" problem associated with the Linux implementation of the POSIX `accept()` system call. We discuss the nature of the problem and how it may affect the scalability of the Linux kernel. In addition, we identify candidate solutions and considerations to keep in mind. Finally, we present a solution and benchmark it, giving a description of the benchmark methodology and the results of the benchmark.

1. Introduction

Offered loads on network servers that use TCP/IP to communicate with their clients are rapidly increasing. A service may elect to create multiple threads or processes to wait for increasing numbers of concurrent incoming connections. By pre-creating these multiple threads, a network server can handle new connections and requests at a faster rate than with a single thread.

In recent years, the term scalability has been used to describe a number of different characteristics, so it may be useful to present our use now. Traditionally, scalability has meant that system performance changes in direct proportion to system resources. For this to be the case, all operations would have to be executed in constant time. Of course, it's impossible to have a system which achieves perfect scalability, but we can certainly try. For our purposes, we will use this interpretation of scalability. We feel that regardless of how many threads are waiting on a socket's wait queue, an `accept()` system call should execute in near-constant time.

In Linux, when multiple threads call `accept()` on the same TCP socket to wait for incoming TCP connections, they are placed into a structure called a wait queue. Wait queues are a linked list of threads that wait for some event. In the Linux 2.2 series kernel, when an incoming TCP connection is accepted, the `wake_up_interruptible()` function is invoked to awaken waiting threads. This function walks the socket's wait queue and awakens everybody. All but one of the threads, however, will put themselves back on the wait queue to wait for the next connection. This unnecessary awakening is commonly referred to as a "thundering herd" problem and creates scalability problems for network server applications.

This report explores the effects of the "thundering herd" problem associated with the `accept()` system call as implemented in the Linux kernel. In the rest of this paper, we discuss the nature of the problem and how it affects the scalability of network server applications running on Linux. We will investigate how other operating systems have dealt with the problem and finally, we will benchmark our solutions. All benchmarks and patches are against the Linux 2.2.14 kernel.

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

This document is Copyright © 2000 by AOL-Netscape, Inc., and by the Regents of the University of Michigan. Trademarked material referenced in this document is copyright by its respective owner.

2. Background

This section is intended to give a detailed view of the current implementation of `accept()` and its problems. It will describe what we found in our initial research and explain the implications of each discovery. For the sake of comparison we will do the same for another widely used operating system, OpenBSD. At the end of the section we will layout the guidelines we used when formulating solutions to the problem.

2.1 Investigation

When a thread wants to listen for an incoming TCP connection, it creates a TCP socket and invokes the `accept()` system call. The system call uses the protocol specific `tcp_accept()` function to do all the work. The relevant sections of this procedure are the manipulations of the thread's state and socket's wait queue. In Linux, a thread's context is represented by a structure (`struct task_struct`) which maintains several variables pertaining to memory allocation and runtime statistics. One of these variables is named `state`. The `state` variable is used as a bitmask to indicate whether a thread is running, sleeping, waiting for an interrupt or yielding to an interrupt. Currently, when a thread calls `accept()` on a TCP socket the thread's state is changed from `TASK_RUNNING` to `TASK_INTERRUPTIBLE` and the thread is placed at the end of the wait queue associated with the socket. At this point, the thread puts itself to sleep and the system resumes normal operation. Every thread accepting on a socket follows this procedure, thus lengthening the wait queue whenever multiple threads `accept()` on the same socket.

The second part of this routine occurs each time another process (local or remote) initiates a TCP connection with the accepting socket. When the connection comes in, the network interface pulls the packet into kernel memory and passes it to the function `tcp_v4_rcv()`. This function parses the TCP packet header and identifies it as an attempt to connect with a listening socket. The TCP stack then calls `wake_up_interruptible()` on the corresponding socket's wait queue to wake and signal a thread to handle the new connection.

To completely understand how the Linux TCP stack awakens threads on a socket's wait queue requires a bit more detail. The socket structure in Linux contains a virtual operations vector that lists six methods (referred to as call-backs in some kernel comments). These

methods initially point to a set of generic functions for all sockets when each socket is created. Each socket protocol family (e.g., TCP) has the option to override these default functions and point the method to a function specific to the protocol family. TCP overrides just one of these methods for TCP sockets. The four most commonly-used socket methods for TCP sockets are:

```
sock->state_change
    (pointer to sock_def_wakeup)
sock->data_ready
    (pointer to sock_def_readable)
sock->write_space
    (pointer to tcp_write_space)
sock->error_report
    (pointer to sock_def_error_report)
```

The code for each one of these methods invokes the `wake_up_interruptible()` function. This means that every time one of these methods is called, tasks could be unnecessarily awakened. In fact, in the `accept()` routine alone, Linux invokes three of these methods, essentially tripling impact of the "thundering herd" problem. The three methods invoked to wake tasks on a socket's wait queue are `tcp_write_space()`, `sock_def_readable()` and `sock_def_wakeup()`, in that order.

Because the most frequently used socket methods all call `wake_up_interruptible()`, the thundering herd problem potentially extends beyond the `accept()` system call and into the rest of the TCP code. In fact, it is rarely necessary for these methods to wake up the entire wait queue. Thus, almost any TCP socket operation could unnecessarily awaken tasks and return them to sleep. This inefficient practice robs valuable CPU cycles from server applications.

2.2 Comparison

In investigating the characteristics of thundering herd issues in Linux, we thought it might be a good idea to see how other systems deal with the issue. In particular, we examined the OpenBSD system to see how it behaves in the `accept()` system call. In OpenBSD 2.6, when a thread calls `accept()` on a socket, the thread puts itself to sleep with a socket specific identifier. When a connection is made to a socket, the kernel wakes up all threads sleeping on that socket's identifier. So it would appear that OpenBSD has the same thundering herd issues as Linux, but this is not the case. The OpenBSD kernel serializes all calls to `accept()`, so only one thread is waiting for a particular socket at any

time. Although this approach prevents the thundering herd condition, it also limits performance, as we will see in section 5.

2.3 Guidelines

When developing solutions to any problem, it is important to establish a few rules to warrant acceptability and quality. While investigating the Linux TCP code, we set forth this particular set of guidelines to ensure the correctness and quality of our solution:

- *Don't break any existing system calls* - If the changes affect the behavior of any other system calls in an unexpected way, then the solution is unacceptable.
- *Preserve "wake everybody" behavior for calls that rely on it* - Some calls may rely on the "wake everybody" behavior of `wake_up_interruptible()`. Without this behavior, they may not conform to POSIX specifications.
- *Make solution as simple as possible* - The more complicated the solution, the more likely it is to break something or have bugs. Also, we want to try to keep the changes as local to the TCP code as possible so other parts of the kernel don't have to worry about tripping over the changed behavior.
- *Try not to change any familiar/expected interfaces unless absolutely necessary* - It would not be a good idea to require an extra flag to an existing function call. Not only would every use of that function have to be changed, but programmers who are used to its interface would have to learn to supply extra arguments.
- *Make the solution general, so it can be used by the entire kernel* - If any other parts of the kernel are experiencing a similar "thundering herd" problem, it may be easily fixed with this same solution instead of having to create a custom solution in other sections of the kernel.

3. Implementation

The fundamental idea behind solving the "thundering herd" problem is to somehow prevent all sleeping threads from waking up. This section will outline the implementation of a couple proposed solutions, including one that was incorporated into the 2.3 development series of the Linux kernel.

3.1 Task Exclusive

One proposed solution to this problem was suggested by the Linux community and incorporated into the 2.3 development kernel series. The idea is to add a flag to the threads state variable, change the handling of wait queues in `wake_up_interruptible()` and implement a new wait queue maintenance method called `add_wait_queue_exclusive()`. To use this solution, the soon to be sleeping thread would set the new `TASK_EXCLUSIVE` flag in the thread structure's state variable, then add itself to the wait queue using `add_wait_queue_exclusive()`. In the case of `accept()`, the protocol specific accept function (`tcp_accept()`) would be responsible for doing this work.

In handling the wait queue, `__wake_up()` (called by `wake_up_interruptible()`) will traverse the wait queue, waking threads as it goes until it runs into its first thread with the `TASK_EXCLUSIVE` flag set. It will wake this thread and then exit, leaving the rest of the queue waiting. To ensure that all threads that are not marked exclusive were awakened, `add_wait_queue()` will add threads to the front of a wait queue, while `add_wait_queue_exclusive()` will add exclusive threads to the end of a wait queue, after all non-exclusive waiters. Programmers are responsible for making sure that all exclusive threads are added to the wait queue with `add_wait_queue_exclusive()`. Special handling is required to wake all exclusive waiters in abnormal situations (like listening sockets being closed unexpectedly).

3.2 Wake One

Another solution, stemming from the idea that the decision point for waking one or many threads should not be made until wake time, was developed here at CITI. Processes or interrupts that awaken threads on a wait queue are generally better able to determine whether they want to awaken one thread or many. This solution does not use a flag in the task structure* and doesn't use any special handling in `add_wait_queue()` or `add_wait_queue_exclusive()`. With respect to the guidelines above, we felt that the easiest way to implement a solution is to add new calls to complement `wake_up()` and `wake_up_interruptible()`. These new calls are `wake_one()` and `wake_one_interruptible()`. They are #defined macros, just like `wake_up()` and `wake_up_interruptible()` and take exactly the same arguments. The only difference is that an extra flag is sent to `__wake_up()` by these macros, telling the system to wake only one thread instead of all of them. This way it's up to the waking thread whether it wants to wake one (e.g., to accept a connection) or wake all (e.g., to tell everyone the socket is closed).

For this "wake one" solution we examined the four most commonly used TCP socket methods and decided which should call `wake_up_interruptible()` and which should call `wake_one_interruptible()`. Where we elected to use `wake_one_interruptible()`, and the method was the default method for all socket protocols, we created a duplicate function just for TCP to be used instead of the default. We did this so the changes would affect only the TCP code, and not affect any other working socket protocols. If at some point later it is decided that `wake_one_interruptible()` should be the generic socket default, then the new TCP specific methods can be eliminated. Based on our interpretation of how each socket method is used, here's what we came up with:

* Although, there is a set of flags passed to `__wake_up()` that resemble the state variable in the task structure, i.e., the flags are set with the same bit masks as those used for the task structure. `TASK_EXCLUSIVE` is still #defined and passed as a bit to `__wake_up()` even though it is not used in the task structure.

```
sock->state_change - (tcp_wakeup)
                    wake_one_interruptible()

sock->data_ready - (tcp_data_ready)
                  wake_one_interruptible()

sock->write_space - (tcp_write_space)
                  wake_one_interruptible()

sock->error_report (sock_def_error_report)
                  wake_one_interruptible()
```

Notice that all three of the methods used in `accept()` call `wake_one_interruptible()` instead of `wake_up_interruptible()` when this solution is applied. The main obstacle with this approach is that system calls like `select()` depend on being awoken every time, even if there are threads ahead of them on the wait queue.

3.3 Always Wake

A third solution, which has not yet been implemented, combines the most desirable characteristics of the two previous solutions. The decision to wake one or many threads would still be deferred until the time of awakening by using `wake_one()` and `wake_one_interruptible()`. However, for the rare case where a thread would always need to wake up (like `select()`), a bit in the threads state could be set to indicate this. These threads would reside at the front of the wait queue and always be awoken on calls to `__wake_up()`. This solution is still easy for programmers to use, and only requires special care for the special cases. It gives the power to decide between awakening one or many threads to the more informed waking thread, while still providing a mechanism for the sleeper to make the decision if it knows better.

4. Performance Evaluation

Our focus is on improving system throughput. In this case, we hope to accomplish our goal by eliminating unnecessary kernel state CPU activity. To measure the performance of each solution we consider two questions. First, how long does it take for all threads to return to the wait queue after a TCP connection is initiated? Second, how does a network service perform under high load/stress situations with the new solutions? We took two different approaches to benchmarking the performance impact of the "wake one" and "task

Threads	Stock	TaskEx	WakeOne
100	4708	649	945
200	11283	630	1138
300	21185	891	813
400	41210	776	1126
500	52144	567	1275
600	75787	1044	599
700	96134	1235	707
800	118339	1368	784
900	149998	1567	1181
1000	177274	1775	843

Table I: The results of the microbenchmark (in usecs) are very rough estimates. But even at such a level of granularity, they still show significant improvement in settle time for the patched kernels over the stock kernel

exclusive” patches. The first is a simple micro-benchmark that is easy to set up and quick to run. We ran this to

get a rough idea of what sort of improvement we can expect with each patch. The other is a large-scale macro-benchmark on the patched kernels, to see if the patch improves performance under high loads as well.

4.1 Small Scale Performance

To measure how much time it takes for all unused threads to return to the wait queue after a connection is made, we wrote a small server program that spins X number of threads and has each of them accept on the same port. We also wrote a small client program that creates a socket and connects to the port on the server Y (in this case 1) times. We issue a `printk()` from the kernel every time a task is put on or removed from the wait queue. After the client has “tapped” the server, we examine the output of the `printk()`’s and identify the points where the connection was first acknowledged (in terms of wait queue activity) and where all threads have returned to the wait queue.

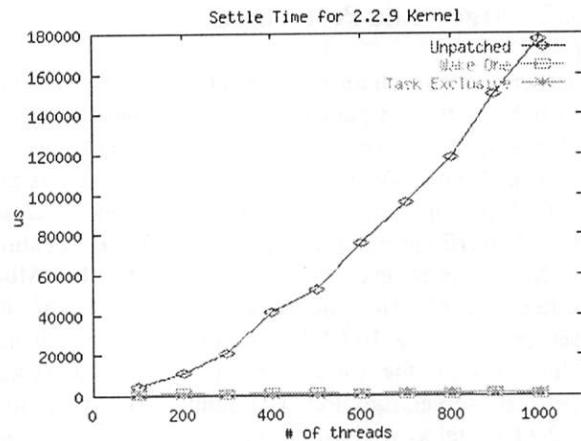


Figure I: This graph shows the difference in the time complexity between the stock kernel and the ones patched with thundering herd solutions.

The results are reported as an estimated elapsed time for the wait queue to settle down after an `accept()` call is processed. The measurements are not exact, as we were using `printk()`s and only ran the tests once. These two points can result in a slight skew of the results in three ways. First, `printk()`’s are not free operations and add to the execution time each time they are used. Second, to provide less room for statistical error, many samples should be taken, but these tests were only run once and could produce slightly different results on subsequent runs. However, even with these degrees of inaccuracy, this micro-benchmark is still able to give us a rough estimation of the time complexity involved with each scenario. Table I gives the settling time for stock and patched kernels with various numbers of threads on the wait queue. The server was running Linux 2.2.9 on a Dell PowerEdge 6300 with four 450 MHz Pentium II Xeon processors, a 100 Mbps Ethernet card and 512M of RAM (lent to the Linux Scalability Project by Intel).

The key observation to be made when looking at these rough estimates is the difference in time complexity. While the stock kernel settles in $O(n)$ time, both of the patched kernels settle in nearly constant time. Figure I illustrates these differences.

4.2 Large Scale Performance

To set up the test harness for this benchmark, the Linux Scalability Project purchased new machines for use as clients against a web server. Four client machines are equipped with AMD K6-2's running at 400 MHz and 100 Mbps Ethernet cards. The server is a four processor Dell PowerEdge 6300 running with 400 MHz Pentium II Xeon processors, 512M of RAM and a 100 Mbps Ethernet card. The clients are all connected to the server through a 100 Mbps Ethernet switch. All machines used in the test are running a 2.2.14 Linux kernel. The server runs Red Hat Linux 6.0 with a stock 2.2.14 kernel as well as the "task exclusive" and "wake one" patched 2.2.14 kernels.

We elected to use the Apache web server as our network service because it's a widely used application and is easily modified to make this test more useful. Stock Apache 1.3.6 uses a locking system on Linux to prevent multiple `httpd` processes from calling `accept()` on the same port at the same time, which is intended to reduce errors and improve performance in production web servers. For our purposes, we want to see how the web serving machine will react when multiple `httpd` processes all call `accept()` at once. We modified Apache so that it doesn't wait to obtain a lock before calling `accept()`. This non-locking behavior is the default on systems where multiple `accept()`s are safe. The patch for this modification can be found on our web page at:

www.citi.umich.edu/projects/linux-scalability

To stress-test our web server, we used a pre-release version of SPEC's SpecWeb99 benchmark, courtesy of Netscape's web server development team. Because the benchmark is pre-release, SPEC rules constrain us from publishing detailed throughput results. However, we can still make general quantitative statements about the performance improvements.

Running the benchmark maintains between 300 and 1000 simultaneous connections to the web server from the client machines and measures throughput by requesting as many web pages as possible. Each connection requests a web page and then dies off while a new connection is generated to take its place. The Apache web server is configured to use 200 `httpd` daemons and does not support keep-alive connections (so idle connections do not linger). All `httpd` daemons accept on the same port. The throughput is measured by SpecWeb99 in terms of how many requests per second each of the 300 to 1000 simultaneous connections can make.

The results of the SpecWeb99 runs are very encouraging. While running with moderate to sizable loads of 300 to 1000 simultaneous connections to the web server, the number of requests serviced per second increased dramatically with both the "wake one" and "task exclusive" patches. While the performance impact is not as powerful as that evidenced by our micro-benchmark, a considerable gain is evident in the testing. The performance increase due to either patch remains steady at just over 50% for all connection rates. There is no discernable difference between the "wake one" and "task exclusive" patches.

5. Application

Up to this point, the evaluation of the elimination of thundering herd problems seems overwhelmingly positive. However, there is one issue that seems unresolved. In the performance testing, SpecWeb99 was run against a modified Apache web server. Why did we put forth the effort to modify our web server and why would anybody want to do so in practice? To answer these questions, we performed a short evaluation of the stock Apache 1.3.9 web server and our patched version.

The stock Apache web server uses various locking schemes to prevent the servers threads from all calling `accept()` at the same time. This is done to prevent internal errors when the server receives connections on many different IP addresses or ports. When running an Apache web server on one IP address and one port, locking around `accept()` is not necessary.

If Apache server threads were all allowed to call `accept()` at the same time, then each thread could process a good portion of the `accept()` system call before a connection is even received. This in turn would reduce the effective overhead of accepting each incoming connection, since half the work is already done. To test this idea, we set up another test against a uniprocessor machine which would show the usefulness of these thundering-herd solutions on more common hardware.

This evaluation used a single processor AMD K6-2 machine running at 400 MHz equipped with a 100 Mbps ethernet card and the same four processor machine described in the macro-benchmark section. The quad-processor was used as a client machine running `httperf` to ensure that the web serving host (and not the client) would be under a significant load. The client was tested using two different configurations: a stock 2.2.14 Linux kernel with a stock locking Apache 1.3.9

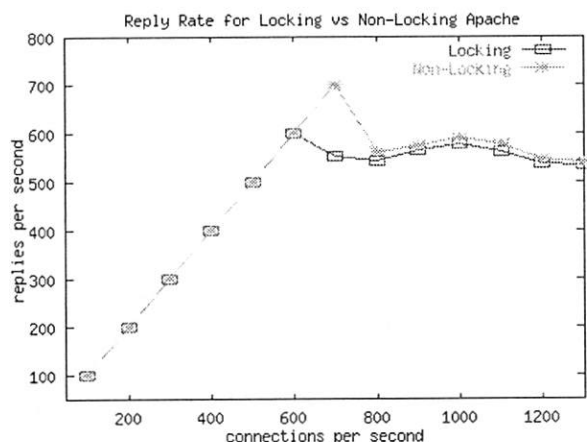


Figure II: This graph shows the rate at which the web server replied (y axis) for each level of client request-rates (x axis). Notice the point at which each server started to lose performance.

web server and the same kernel with the modified non-locking Apache 1.3.9 web server. The Apache web servers were configured to run a modest 20 serving threads (httpd's) and to not support keep-alive connections.

The results of this test are plotted in Figure II. This graph demonstrates how Apache can increase the threshold rate at which it begins to fail by having all 20 httpd's accept at the same time, rather than deferring the accept overhead until later. You can imagine that if more httpd's are started the difference in thresholds would decline, because on a stock 2.2.14 Linux kernel the system would begin to feel the effects of the thundering herd problem. It is not uncommon though, for medium to high traffic sites run more than 100 httpd processes.

6. Conclusion

By thoroughly studying this "thundering herd" problem, we have shown that it is indeed a bottleneck in high-load server performance, and that fixing it significantly improves the performance of a high-load server regardless of the method used. This performance increase is due to the fact that less time is spent in the kernel needlessly scheduling tasks which are not yet ready to run. All solutions presented resolve the issue by awakening as few tasks as necessary, thus reducing kernel overhead.

At first look, the "task exclusive" solution appears to be fairly complex. Upon closer examination though, it seems to fit in well with the new structure of Linux wait queues (doubly linked in 2.3 to make end-of-queue additions fast). Extra demands are placed on the programmer to get this solution to work, but the fix is extensible to all parts of the kernel and appears not to break any existing system calls. The "wake one" solution, on the other hand, is cleaner, easier for programmers to implement and is also extensible to all parts of the kernel. This fix is easily used by programmers since it requires just one line of code.

As previously mentioned, the process that awakens tasks is usually better able to determine if it wants to awaken one or more tasks. However, in the case of `select()`, the selecting process will want to be awakened regardless of whether or not it will continue on to handle the connection (perhaps it is monitoring the socket and collecting some statistics). For this case, the "task exclusive" model is a better fit. Conversely, if an application error occurs, a program may like to inform all of its associated tasks which are waiting on a socket. For this case, the "wake one" model is the better fit. Perhaps the most sound and elegant solution is the "always wake" hybrid of these two solutions which was presented in section 3.3.

6.1 Availability

All work and patches presented and used in this paper were written and performed at CITI and are available on the Linux Scalability Project's home page at <http://www.citi.umich.edu/projects/linux-scalability/>

6.2 Acknowledgements

Many Linux developers have contributed directly and indirectly to this effort. The authors are particularly grateful for input and contributions from Linus Torvalds and Andrea Arcangeli. Special thanks go to Dr. Charles Antonelli and Professor Gary Tyson for providing hardware used in the test harness for this report. The authors would also like to thank Peter Honeyman and Stephen Tweedie for their guidance, as well as the USENIX reviewers for their comments.

7. References

- [1] M Beck, H Bohme, M Dziadzka, U Kunitz, R Magnus, D Verworner, *Linux Kernel Internals*, 2nd Ed., Addison-Wesley, 1998
- [2] Samuel J Leffler, Marshall K McKusick, Micheal J Karels, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989
- [3] Stevens, W Richard, *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI*, 2nd Ed., Prentice-Hall, Inc., 1998
- [4] The Single UNIX Specification, Version 2, www.opengroup.org/onlinepubs/7908799
- [5] Apache Server, The Apache Software Foundation. www.apache.org
- [6] D. Mosberger and T. Jin, "httperf – A Tool for Measuring Web Server Performance," *SIGMETRICS Workshop on Internet Server Performance*, June 1998.
- [7] SPECWeb99, Standard Performance Evaluation Corporation. www.spec.org
- [8] Apache Performance Tuning, The Apache Software Foundation. www.apache.org/docs/misc/perf-tuning.html

Permanent Web Publishing

David S. H. Rosenthal
Sun Microsystems Laboratories
Vicky Reich
Stanford University Libraries

"Diffused knowledge immortalizes itself"
Vindiciae Gallicae Sir James Mackintosh (1765-1832)

Abstract

LOCKSS (Lots Of Copies Keep Stuff Safe) is a prototype of a system to preserve access to scientific journals published on the Web. It is a majority-voting fault-tolerant system that, unlike normal systems, has far more replicas than would be required just to survive the anticipated failures. We are exploring techniques that exploit the surplus of replicas to permit a much looser form of coordination between them than conventional fault-tolerant technology would require.

1. Introduction

In a classic paper Leslie Lamport^[lamport] set out the basis for building a distributed system with n replicas which could tolerate f faults, where $f = \lfloor (n-1)/3 \rfloor$, by having the replicas vote in "elections" to decide the system's behavior.

We describe work in progress to develop a solution to an important real-world problem, the preservation for future generations of scientific, technical and medical (STM) journals published on the web. Any system guaranteeing long-term preservation of information must tolerate faults such as disk crashes, network outages and malicious attacks. Certain special characteristics of this problem mean that in our case n is much greater than $3f+1$. The large number of replicas allows us to try a somewhat different approach to fault-tolerance. If the classic approach is analogous to elections, our approach is analogous to opinion polls.

We start by outlining the problem and generating requirements for a solution. We show how these requirements led to a design for which $n \gg 3f+1$. We describe the design in some detail and provide a brief report on the status of the implementation. We compare our approach with related work and assess its strengths and weaknesses. We conclude by speculating about the usefulness of similar techniques to other applications for which large numbers of replicas are appropriate.

2. The Problem

In most respects the Web is a far more effective medium for scientific, technical and medical (STM) communication than paper. Stanford Library's Highwire Press^[highwire] led the transition of STM publishing from paper to the Web and now publishes the on-line editions of about 180 of the top STM journals. They pioneered techniques such as datasets in spreadsheets behind graphs, dynamic lists of citing papers, e-mail notification of citing papers and so on. These, added to the basic hyperlinks and searchability, make the Web versions both easier to access and more useful when accessed than the paper ones. Web versions frequently appear earlier and contain much more information. Many journals now publish some papers only on the Web.

Librarians have a well-founded confidence in their ability to provide their readers with access to material published on paper, even if it is centuries old. Preservation is a by-product of the need to scatter copies around to provide access. Librarians have an equally well-founded skepticism about their ability to do the same for material published in electronic form. Preservation is totally at the whim of the publisher.

A subscription to a paper journal provides the library with an archival copy of the content. Subscribing to a Web journal rents access to the publisher's copy. The publisher may promise "perpetual access", but there is no business model to support the promise. Recent events have demonstrated that major journals may vanish from the Web at a few months notice.

This poses a problem for librarians, who subscribe to these journals in order to provide both current and future readers with access to the material. Current readers need the Web editions. Future readers need paper; there is no other way to be sure the material will survive.

The transition to the Web will not be complete until librarians are willing to buy Web-only subscriptions. To do so they need confidence in their ability to provide their readers with long-term access to the content they are buying. The problem is in three parts:

- The bits themselves must be preserved. All digital storage media have a limited lifetime; the bits need to migrate from one medium to another over time. In practice it is difficult to fund a bulk copying effort when a medium starts decaying, so all but the most valuable bits are lost at each transition^[bit-rot].
- Access to the bits must be preserved. Suppose a reader clicks on a link 20 years from now that doesn't resolve because the only copy is now on a CD in a secure store. Whom does the reader call to get the CD from the store into a specially preserved CD drive? Unless links to pages continue to resolve, the material will effectively be lost because no-one will have the knowledge or patience to retrieve it.
- The ability to parse the bits, once accessed, into human-readable form must be preserved^[format].

It is important to observe that there can be no single solution to this problem. A single solution of itself would be perceived as vulnerable. By proposing one solution we are not arguing that other solutions should not be developed and deployed. Diversity is essential to successful preservation.

3. Requirements

Librarians' technique for preserving access to material published on paper has been honed over the years since 415AD, when much of the world's literature was lost in the destruction of the Library of Alexandria^[alexandria]. Their method may be summarized as:

Acquire lots of copies. Scatter them around the world so that it is easy to find some of them and hard to find all of them. Lend or copy your copies when other librarians need them.

In this context, note the distinction between archives, which we are not discussing, and general circulating collections, which we are:

- The goal of an archive is preservation, typically of material that is unique and/or impossible to replicate widely. Access is restricted via locked

stacks, access logs and so on to ensure preservation.

- The goal of a circulating collection is access, typically to replicas of material that is widely copied. Risks are taken with preservation to achieve access. Copies are on open shelves and are loaned to readers on a promise that they will eventually be returned.

Libraries' circulating collections form a model fault-tolerant distributed system. It is highly replicated, and exploits this to deliver a service that is far more reliable than any individual component. There is no single point of failure, no central control to be subverted. There is a low degree of policy coherence between the replicas, and thus low systemic risk. The desired behavior of the system as a whole emerges as the participants take actions in their own local interests and cooperate in ad-hoc, informal ways with other participants.

If librarians are to have confidence in an electronic system, it will help if the system works in a familiar way. The fundamental requirement for LOCKSS (Lots Of Copies, Keep Stuff Safe)^[lockss] was, therefore, to model their techniques as closely as possible for material published on the Web.

If libraries can take physical custody of the journals they purchase, in a form that preserves access for their readers, they can assume the responsibility for their future. If a library takes custody of a copy of the Web journal, the copy can behave as a Web cache and provide access whether or not it is available from the original publisher. If many libraries do so, the caches can communicate with each other to increase the reliability and availability of the service, as inter-library loan increases the reliability and availability of access to information on paper.

Another way of looking at a system of spreading copies of Web journals around the world is that the librarians running the system are buying insurance for their journal subscriptions. What are the librarians insuring against? Reasons why their readers would lose access to a journal include:

- Failure to renew a subscription, for example because of budget cuts or a price increase by the publisher.
- Change of policy by the publisher, for example because a not-for-profit journal was taken over by a for-profit publisher.

- A publisher going out of business.
- Incompetent or careless management of the publisher's web service.

In all these cases the symptoms are either a refusal by DNS to resolve the name in the link URL or a refusal by the server named to supply the content.

Libraries have to trade off the cost of preserving access to old material against the cost of acquiring new material. They tend to favor acquiring new material. To be effective, subscription insurance must cost much less than the subscription itself.

The biggest journal Highwire publishes generates about 6GB/year. A cheap PC to hold 5 years' worth might cost \$600 today, which is about 10% of the subscription for the 5 years. If the running costs of the system can be kept low enough, it should now be practical for many libraries to maintain their own copies. The prospects for this insurance improve as equipment prices fall and subscription prices rise^[disk-cost].

In this context, Open Source development is crucial:

- The goal of the project is to inspire confidence. It is hard to have well-founded confidence in a system whose operations are kept secret.
- The system's economics mandate free distribution of the software; there's barely a budget for the hardware.
- The longevity of the system will require many generations of programmers to refine it as problems are encountered.

4. Implications

Conventional replicated fault-tolerant systems are designed around the question "how few replicas are needed to survive the anticipated failures?" This required number of replicas, perhaps 5, are then organized into a tightly administered system.

To solve our problem, each library wanting to insure their subscription must take custody of a copy of the Web journal in question. If the system is successful there will be many more replicas than needed to survive the expected failures. There may be hundreds. It isn't possible to administer hundreds of systems, each under the control of a separate institution, as tightly as the small number of replicas under centralized

administration in a conventional fault tolerant system. Nor is it efficient to have hundreds of systems participate in every operation of a distributed system in order to survive the failure of a tiny fraction of them.

In the real world there is no authority controlling libraries. Anyone can claim to be a library. Individual librarians assess the credibility of such claims on the basis of experience. They are suspicious of the long-term dependability even of neighboring libraries whose *bona fides* are not in question. Creating tightly controlled long-term cooperative efforts in this environment is not effective. LOCKSS therefore needs to trade off its surplus of replicas for a looser form of cooperation among the replicas.

5. Design

The design goal for LOCKSS is to provide librarians with a cheap and easy way of running Web caches which pre-load the journals they subscribe to, preserve them for posterity by never flushing the cache, and serve their pages to the library's readers if the publisher does not. The design falls into three parts:

- Pre-loading the cache as new issues of the journals are published.
- Ensuring that readers can access the journals from the publisher or from the cache.
- Preserving the contents of the cache.

The design takes advantage of some unusual features of the Web editions of STM journals:

- The peer-review system requires that the articles are immutable once published. The history of publishing on paper has reinforced this feature.
- The web site for a journal has a logical structure, with directories for each volume, each issue within the volume, articles within the issue, and so on.
- New content is published on a fairly regular cycle that may be weekly or monthly, not hourly.

It should be emphasized that we are not designing a general-purpose Web content preservation system. LOCKSS is designed to preserve only journals published by Highwire; we are in control of both ends of the process and could if necessary alter the HTML Highwire generates to make preservation easier. It may be possible to apply the system to other types of

content, but that is not at present a design goal. LOCKSS is clearly not suitable for volatile content.

It should also be emphasized that we are describing work in progress. As this paper is being written we are preparing the prototype for the system's first major test, using a small group of libraries and a single journal. We expect the design to evolve as we gain experience.

5.1. Collecting

A librarian instructs an instance of LOCKSS to preserve a volume of a journal by providing the publisher's root URL for the volume and a frequency of publication, say monthly. At that frequency a web-crawler starts from the root URL and fetches all new pages within that sub-tree. The publisher's web server sees this access as coming from an authorized IP address, so it is allowed. Note that we don't depend on readers accessing the material to populate the cache.

This is off-the-shelf technology; we currently use the *w3mir* crawler^[w3mir], which is written in Perl and easy to adapt to our needs. The only change needed was an interlock with the cache preservation code, to prevent content being checked while it was being collected.

5.2. Serving

This is also off-the-shelf technology. The prototype uses the Apache^[apache] web server to export the contents of each cache to the local network's users.

Tighter integration with the cache management code will be needed in the future to support the code that prevents the publisher's access control system being subverted [see 6.3.2]. At that point we expect to switch to a simple HTTP server in Java, which can check the cache manager's internal data structures to determine if accesses are appropriate.

5.3. Preserving

The heart of LOCKSS is the process by which caches cooperate to detect and repair damage. The caches communicate using an IP multicast protocol to discover which URLs should exist and what their contents should be. This protocol runs continually but very slowly between all the caches. If a cache discovers a missing or damaged URL it can fetch a new copy via HTTP from the original publisher, or from one of the other caches. Care has to be taken not to subvert the publisher's access control mechanism; content should only be delivered to sites that have rights to it.

The process works this way. A cache will notice that a part of the material it is preserving has gone long enough without being checked. It will multicast a call for a *poll* to decide the value of the message digest of the sub-tree below the directory representing that part. Other caches hearing the call will compute their digests and reply. The caches hearing the replies will tally the poll. If they are on the winning side their cache is intact. If they are on the losing side, their cache contains some damage. The damage is located by:

- Calling a poll to determine the set of names in the directory representing the part containing damage.
- Calling a poll on the message digest of the sub-tree below each name.

The process descends the tree until it finds files instead of directories. If the digests agree the file is intact. If not the file is damaged and a new copy is fetched¹. The enumeration of the names in the directories also finds any extra names, which are removed, and any missing names, which cause the corresponding file or sub-tree to be fetched.

6. Protocol

We describe the protocol from the bottom up, starting with the basic polling mechanism, then the different types of polls, and then the policies that string the polls together.

6.1. Elections & Polls

We call the inter-cache protocol LCAP [Library Cache Auditing Protocol]. The design was inspired by Scalable Reliable Multicast (SRM)^[srm], a peer-to-peer reliable multicast protocol whose importance for LCAP is its use of random timeouts to run a "sloppy" form of election.

All participants in SRM subscribe to and multicast packets to a single IP group address. If a missing packet is detected, a request is multicast for the packet to be re-transmitted. If a request is received for re-transmission of a packet that has been received a random timer is started. If a re-transmission of that packet is received while the timer is running, the timer is cancelled. Otherwise when the timer expires the packet is re-transmitted to the group address.

¹ In production, we'll be more careful. Files won't be overwritten or removed but moved to a backup tree.

This random timeout mechanism is used to elect a participant to perform the re-transmission. It is not a perfect mechanism; sometimes failing to elect anyone and sometimes electing more than one. If no one is elected, the requester times out and tries again. If more than one is elected, the repair is re-transmitted more than once, wasting bandwidth but doing no other harm.

In practice the "sloppy election" mechanism is very effective in those cases where delays several times the length of a packet round-trip are tolerable. It is statistically very likely to elect a single participant. It has built-in load balancing, electing a participant at random. It tolerates faults; if the participant with the shortest random delay fails before transmitting its repair another will take its place. For our purposes these multicast, random timeout based "sloppy elections" have many attractive features:

- Each election selects its own electorate; there is no configuration database to maintain in a consistent state.
- They load-balance automatically. If there are more than enough potential voters the actual voters will be chosen at random.
- The elections survive lost packets very well.

In the real world elections are tightly controlled and very expensive procedures. A register of electors must be maintained, with rigorous procedures for qualifying those who may vote. Voters have to identify themselves at the polls. Long experience has led to many precautions against fraud.

Opinion polls have none of this overhead yet almost always predict the result of an election correctly. They do require careful attention to sampling and question design, but because they don't need the administrative structures nor the mass participation of a real election they are much cheaper and quicker.

LOCKSS uses a variant of the SRM "sloppy election" technique to implement something akin to an opinion poll. The caller of a poll announces:

- the *subject* of the poll, the URL to which it applies, and
- a *hurdle* for the poll, the number of agreeing votes needed to make it valid, and

- a *duration* for the poll, setting the time that will elapse before the votes will be tallied, and
- a *challenge* for the poll, a random string that is prepended to the data to be digested.

These values appear in the header of each packet sent as part of a poll.

A participant receiving a call chooses a random delay in the duration when it plans to vote. The voter chooses a random *verifier* string, prepends the challenge and the verifier to the data, and computes a message digest. When the timer expires it multicasts the remaining duration, the challenge, the verifier and the message digest. The challenge and the verifier prevent replays and force each voter to prove that they have the content in question at the time of the poll.

Votes are tallied at each participant receiving them by prepending the challenge and the verifier from the vote to the local copy of the data and computing the message digest. If this digest matches the one in the vote it is an agreeing vote, otherwise it is a disagreeing vote.

A participant receiving more than the hurdle number of agreeing votes before their timer expires can, if the agreeing votes are the majority, decide that there is no need to vote and cancel the timer². In determining that it agrees with the majority it will have checked its local copy. There's no harm in voting unnecessarily, but participants need not tally excess votes.

There's no "electoral register" determining who can and cannot vote. If a cache has a copy of the data in question it can vote, because it is doing the job of preserving access to the data. This models the real world; there is no authority deciding who is qualified to be a library.

6.2. Operations

The cache consistency checking process uses two different types of polls:

- A *compare* poll asks voters to compute the message digest of the challenge, their verifier and the data in the sub-tree below the subject URL.

² In practice the timer is not cancelled but suspended, in case of a late rush of disagreeing votes, see 7.2.4.

- An *expand* poll asks voters to compute the message digest of the challenge, their verifier and the set of names in the directory named by the subject URL. Voters in an expand poll also send the set of names in their directory.

A participant tallying a compare poll will either:

- Agree with the majority, in which case nothing need be done. It has been established that the system as a whole is storing at least the hurdle number of good copies.
- Disagree with the majority, in which case part or all of the local copy of the sub-tree named by the subject is bad. The participant chooses a random timeout and, if another participant has not already done so, calls an expand poll on the subject URL.

A participant tallying an expand poll counts the number of votes for each set of names. The set with the most votes, provided it reaches the hurdle, is the winner. The participant then compares the winning set with their set:

- Names in the local set but not in the winning set need to be removed.
- Names in the winning set but not in the local set need to be fetched.
- Names in both sets need to be compared. For each such name the participant chooses a random timeout. When it expires, provided some other participant has not already done so, the participant calls a compare poll on the name.

In this way the checking process walks the directory tree to locate and repair damage.

6.3. Policies

The system depends on a number of parameterized policies. Over time, our experience with the system will determine if the current set of policies and the corresponding parameter values are adequate.

6.3.1. Rate Limits & Poll Durations

Before we started implementing LOCKSS we expected it to run very slowly just because there was no need for speed. Two things changed this:

- Voting in a top-level poll takes a long time simply because top-level polls typically check an entire issue of a journal, which will range between a few hundred megabytes to a few gigabytes. The voter needs to compute the message digest of all this data once for its own vote, and once for each other vote it checks. The data will normally be hashed about the hurdle number of times. The obsolete 100-200MHz PCs we're using can take an hour or two to do this for our test journal.
- Integrity considerations [see 7.2.2] imply running as slowly as possible is important. The system must run fast enough to compare cached data on average several times between losses, but no faster.

While the need to compute message digests makes top-level polls take a long time, polls checking an individual file could happen much more quickly. The caller of a poll decides on a duration by measuring how long it takes to compute its own digest, multiplying by the hurdle number it chooses, and by a safety factor. The system limits bandwidth consumption by imposing a minimum duration for polls.

6.3.2. Access Control

Each time a cache votes on the winning side it is demonstrating that it has a good copy of the sub-tree in question. Caches remember the winning votes they hear in compare polls for some time, longer than the expected time between failures. They will provide repairs only to caches they remember having voted on the winning side in a poll for the corresponding sub-tree.

In this way they avoid subverting access control. The only way to get a copy other than from the publisher is to have shown in the recent past that you used to have a copy. At some point this recursion arrives at a copy that came from the publisher, and thus satisfied at that time the publisher's access control restrictions.

6.3.3. Maintaining Redundancy

If polls of a given sub-tree consistently fail to reach the hurdle number, this signifies that there aren't enough copies being preserved and the content is at risk. At this stage we believe it is appropriate to notify the people running the system, who can arrange for more copies to be created. At a later stage it might be possible to have LOCKSS instances keep a reserve of disk space in which they can store copies of at-risk material while people cope with the situation.

7. Integrity

So far, we have described a system in an ideal world without malicious participants. Alas, even libraries have enemies^[enemies]. Governments and corporations have tried to rewrite history. Ideological zealots have tried to suppress research of which they disapprove. We have to assume that bad guys will try to subvert LOCKSS if it gets deployed.

7.1. Conventional Approaches

One approach to preventing the bad guy rewriting history that is often suggested is to have the publishers sign the articles they publish. Readers could then check the signature to determine that the document had not been modified since it was signed. We encourage publishers to sign articles (they don't at present) but even if they did it wouldn't guarantee preservation:

- For the future reader to get an article, access to it has to be preserved. Signing the articles helps to verify a copy as authentic once one is obtained, but doesn't help to get a copy in the first place.
- For the future reader, or cache, to be able to check the signature on an article and verify that their copy is authentic they have to obtain the publisher's certificate, check that a trusted Certificate Authority has signed it, and that it hasn't been revoked. This reduces the problem of preserving access to information under the librarians' control (the articles) to the problem of preserving access to smaller units of information outside the librarians' control (the certificates). There's no guarantee that the Certificate Authority will survive the many decades of our timescale.
- The validity of the signature depends on the publisher's private key being kept secret, and on the encryption and hash algorithms involved remaining unbroken. Neither is very likely over our timescale.

Although certificates and signatures are not useful over the long term it is certainly possible to design a system for preserving access in which they are used only over limited periods of time.

Suppose an institution is established which decides from time to time which other institutions are eligible and competent to take part in the preservation effort. It would maintain a registry of certificates a participant could use to decrypt and validate communications from

other participants. This system need not use multicast communication, it would have a rendezvous point similar to the Service Location Protocol's daemon^[slp]. Participants would use this to locate one another. As institutions were subverted or failed to maintain adequate standards of preservation the registry would be updated to delete or revoke their certificates.

Again, the problem of preserving access to information under the librarians' control has been reduced to the problem of preserving smaller units of information outside the librarians' control. Only this time the design has a single point of failure, the registry. Once the bad guy has subverted the registry the entire system is compromised.

The fundamental weakness of conventional approaches is that they divide guys into good and bad by having some external authority place white or black hats on them. A guy with a white hat is free to subvert the system. Just because an institution is a library, it doesn't mean that everyone will, or should trust it. A university library in Greece might, for example, regard a university library in Turkey, as a reliable source of information about chemistry but unreliable on the topics of Aegean geography or Kurdish history. Equally, just because some years ago a library was trustworthy, that doesn't mean they're trustworthy now. The government funding the library may have been replaced with a less scrupulous one.

7.2. Alternative Approach

The approach we're experimenting with in LOCKSS is to divide guys into good and bad by observing and remembering their behavior over a period of time. For our purposes, a good guy is one who:

- maintains a good copy of the journal content,
- votes in polls to prove that the copy is good and to help others prove that their copies are good too,
- remembers that others have voted in the majority for a long time,
- and supplies good copies to others when requested to repair damage.

A bad guy is one who, among many potential crimes:

- votes too early or too often,
- votes on the losing side of too many polls,

- fails to verify their vote on request,
- or supplies bad copies to others.

Note that these are all public actions, observed by others.

7.2.1. Maintaining a Reputation

It's not important for the functioning of LOCKSS that a participant actually be a library, only that the participant exhibit the behavior of a library over a period of time. Because LCAP is a peer-to-peer multicast protocol, the behavior of each participant is visible to the others. They can observe, remember and make their own estimates of the participant's reliability.

This system has many analogies to "reputation mechanisms" in on-line game environments like Ultima Online^[ultima]. Just as in our case, there's no way of knowing "who a participant really is". Indeed, part of the attraction of the game is that players can experiment with multiple identities. The avatar of a particular identity carries a reputation based on its recent actions as observed by other avatars. A bad reputation can be cleaned up over time by performing actions others judge as laudable. Note, however, that games maintain a central registry of avatars' reputations. In the LOCKSS model there is no central registry, each "player" maintains its own registry of other players' reputations.

7.2.2. Running Slowly

Recent actions have higher weight in determining credibility than ancient ones, reflecting the fact that bad guys can reform, and that good guys can be subverted. The latter observation leads to the interesting conclusion that the system must be designed to run extremely slowly, both in the sense of wall-clock time, and the number of operations needed to make a significant change to the cached information:

- Running very slowly limits the damage that a guy who turns bad can do while he retains a reputation based on his actions before he was subverted.
- Running very slowly means that in order to do significant damage a guy must persist in taking bad actions over a long period of time. If a bad guy calls your telephone, your goal should be to prolong the conversation. This allows law enforcement to track the offender down. In our case the system requires the bad guy to take actions

that are both public and obviously bad over a long period of time, allowing the good guys time for location and dissuasion. No system can be immune from penetration; systems should be designed to slow the bad guy's progress and limit the potential damage.

- It is easy to mount a denial of service attack against any multicast protocol. Forcing the protocol to run very slowly makes these attacks unattractive to the bad guy. Sending lots of bogus packets to the LOCKSS IP multicast group address will cause polls underway to fail to reach the hurdle and prevent new ones being called. But as soon as the flood stops the system reverts to normal operation. Preventing the system achieving its goal of long-term preservation requires the bad guy to flood the group for years on end.

7.2.3. Detecting Bad Guys

If no bad guys are active and no participants are losing data, top-level compare polls will be taking place infrequently, and each will be a landslide. To maintain credibility, participants must vote on the winning side in these polls. Doing so is hard work; it requires the voter to compute message digests of hundreds of megabytes of data several times over. This is a good thing:

- It achieves the goal of making the system run slowly in wall-clock terms.
- It provides the system with some inertia and requires the bad guy to invest a lot of effort before he can make an impact on its behavior.
- It makes sure each participant's reputation information is up-to-date. This allows the credibility of inactive or subverted participants to decay quickly, limiting the amount of damage subversion can do.

If no bad guys are active but occasionally participants lose data, polls will descend the tree on occasion and some will have a few dissenting votes. This is the normal behavior of the system.

The structure of a typical journal web site places many intermediate directories between the root and the actual journal articles. It takes many polls to descend from the root to an actual article the bad guy might target. If polls are observed close to the leaves of the tree in which the result is close, or even where there are

substantial numbers of dissenters, one or more bad guys are active. The people running the participants can take measures to stamp them out.

7.2.4. Preventing Fraud

Each participant in a poll verifies a proportion of the votes they tally to help prevent fraud. Very few votes are verified if the poll is a landslide; the proportion rises as the poll becomes an even contest.

The verifier in the vote is actually the digest of a random string that the voter keeps private. The tallier verifies the vote by unicasting a verification request to the sending address in the vote naming the subject and the challenge. The voter replies with the subject, the challenge and a string whose message digest is the verifier in their vote. Voters failing to verify one of their votes after several attempts lose credibility quickly; it is likely that they are being spoofed.

7.2.5. Advantages

This system is strong in some unusual ways:

- There is no central coordination point that can be attacked. Each participant is independent; acting in its own interests, trusting others only as far as necessary and no further than experience shows them to deserve trust. The design goal is that the only way to subvert the system would be to subvert a majority of the participants.
- The system makes as few demands on the infrastructure as possible. It doesn't depend on services such as the Domain Name System, or a Public Key Infrastructure or some mythical Library Certification Organization. All that's needed is for the underlying network to route IP unicast and multicast datagrams.
- It doesn't depend on preserving any meta-information. Provided enough participants preserve the journal articles themselves, a site can corrupt or lose any or all of its information. The more it does, the less its credibility will be among the other participants for a while.
- It doesn't depend on keeping anything secret for any length of time, especially not passwords or encryption keys. Voters need to keep the string that generated their verifier secret for the length of the poll, but this is all.
- It doesn't depend on encryption or hash algorithms resisting attack, because it doesn't use encryption. It does use a hash algorithm during a poll, but this need resist attack only for the duration of enough polls to build or destroy a reputation.
- By operating slowly even on human timescales the system makes it easier to detect an attacker and limits the damage he can do before being stopped.

8. Implementation

The prototype's implementation of the LCAP protocol is in Java. It makes heavy use of threads to maintain the context for ongoing polls, and to ensure that time-consuming operations like computing the message digest of a few hundred megabytes of journal data don't interfere with other tasks. The source will be released under a Stanford equivalent of the U.C. Berkeley license.

It uses SHA-1^[digest] as the message digest function, combined with a filter that parses the HTML of an article to isolate the part that represents the text the authors wrote. This is necessary because successive fetches of a given article from Highwire do not return exactly the same bytes:

- Some journals place advertisements on their pages; the advertising system selects different ads at different times.
- Some features of the article presentation, such as the list of citing articles, change over time.

The current filter is rather crude, more sophisticated versions will be needed before the system goes into production.

9. Production Use

When LOCKSS gets into production, librarians will have to install new instances, manage them through their useful life and replace them when they fail or fill up.

9.1. Installation

For the prototype, we are developing an installation process based on the Linux Router Project's^[lrp] distribution. The librarian will download the image of a generic boot floppy disk, boot it, answer a few configuration questions and then choose an option that re-writes the floppy disk into a configured boot floppy

for the new system. This will be write-locked and used to boot the system in production.

Each time the system boots it will start with a clean, known-good system image in RAM-disk. It will then download, install and run the LOCKSS code. The only data on the hard disk that survives across reboots will be the cache contents and meta-data.

9.2. Management

One major goal of the initial tests is to provide the information needed to design a management interface for the system. We don't yet understand what librarians will need in order to understand and have confidence in the normal operation of the system, nor to detect and respond to abnormal events.

9.3. Replacement

When a LOCKSS instance fails or fills up it can simply be replaced by a new, empty instance assigned to the same journal. The new instance will detect the missing data and reload it from the publisher or other caches. To avoid wasting time and bandwidth, we expect to provide a "clone" option that would allow the librarian to nominate an existing instance from which the new instance's cache would be copied.

10. Performance

There are three important performance metrics for LOCKSS once it is deployed in production:

- What does it cost a library to run it?
- How often does the system as a whole lose or corrupt journal articles?
- What is the probability that a reader will encounter a missing or corrupt article?

Credible numbers for these metrics will not be available for many years. The best we can do right now is some back-of-the-envelope estimates of the I/O, bandwidth and failure rates. These encouraged us to go ahead with the alpha test, but are too sketchy to publish. We expect to report measurements from the alpha test when we present this paper.

11. Related Work

11.1. Fault Tolerance

The conventional approach to fault tolerance through a limited number of replicas is brilliantly illustrated by Miguel Castro & Barbara Liskov^[castro], who built a replicated, fault-tolerant implementation of NFS that benchmarked only 3% slower than the baseline implementation when no failures were encountered and, of course, infinitely faster when they were.

11.2. Internet Archive

LOCKSS is not an archive, and it does not attempt to preserve general Web content. An ambitious attempt to archive the entire Web is underway at the Internet Archive^[archive]. They have currently collected almost 15TB of data, which is primarily stored in a tape robot. As an archive, their mission is primarily preservation, which they plan to ensure by careful treatment of stored data and media migration, not replication. They do not attempt to ensure that the original URLs continue to resolve.

11.3. Intermemory

In the opposite direction, a team at NEC's Princeton labs built a replicated, distributed Internet-scale file system^[nec]. Machines joining the system volunteer disk space to the file store, which uses hashing techniques to smear stored information across multiple replicas. This preserves access to files via the names assigned to them when they are stored, and to their contents via replication. This Intermemory system shares with LOCKSS the basic approach to preservation through replication and copying among unreliable storage systems, but differs in that it exports a file system interface rather than a Web interface, and that its internal workings are obscure to the uninitiated.

11.4. Digital Library research

The NSF is coordinating a major research initiative into the general problem of constructing a Digital library^[dlr]. Projects funded by this DLI2 initiative address a much broader set of issues than LOCKSS, including versioning documents as they change, a vast range of protocols and formats not just HTTP/HTML, and issues around metadata. Because their problem is much harder their technology is not yet deployable.

11.5. Robust URLs

Thomas Phelps and Robert Wilensky^[words] at U.C. Berkeley have discovered that a Web document can be found uniquely with very high probability if a surprisingly small number, from 5 to 8, of carefully

chosen words from the document are given to a search engine. They propose that links to documents be augmented with these signature words to provide browsers with a viable fallback if the URL fails to resolve. This is an interesting idea, but it assumes that the document is somewhere accessible to the search engine after its original publisher has failed, and that the search engine has permission to read it.

This insight could usefully be combined with ideas from Freenet^[freenet], a distributed, search-based information store. Freenet shares with LOCKSS the goal of a system free of the vulnerabilities of central administration and control, but it does not attempt to preserve information whose value is not related to its popularity, and each server appears to trust every other server to supply authentic copies of data being stored.

11.6. Napster

Napster^[napster] provides an interesting example of combining many replicas of a single data item, in their case a song in MP3 form, to form a highly available data resource. Of course, the Napster directory service is itself a single point of failure. The Gnutella distributed directory service would have been more relevant to our problem.

12. Assessment

We stated three goals for LOCKSS. How does the design rate against them?

- The bits must be preserved. If enough replicas can be deployed the system should have a very low probability of losing bits accidentally. The system's effectiveness at preventing malicious actions destroying bits is open to debate. It may be necessary to use encryption and to identify and authorize the participants.
- Access to the bits must be preserved. Readers in participating institutions should have a high probability of having their original links resolve to good copies of articles.
- The ability to parse the bits into human-readable form must be preserved. The process of continual gradual replacement of the software, driven by the need to replace the hardware as it breaks or fills up, allows for format conversion as it becomes necessary.

13. Future Work

We're running an initial test of the prototype for a couple of months with about a dozen instances and a single journal starting in April 2000. We plan to assess this test, incorporate the experience and run a second test at a much more realistic scale later in the year. We hope this test will include an attack team trying to subvert the system.

We're also exploring the suitability of LOCKSS for applications other than journals. One obvious example is the government documents that used to be kept on paper in the "depository library" system, but which are now being published on the Web.

Broader applications of the underlying model of fault tolerance through massive replication and "sloppy" elections are harder to see. LOCKSS as an application has many unusual characteristics. Nevertheless, we remain convinced that there is something fundamentally interesting in the idea of a system based on multicast protocols in which all actions are public and participants can make their own independent assessments of each other's credibility.

One valid criticism of LOCKSS is that all monocultures are vulnerable, and if deployed *en masse* LOCKSS would be a monoculture. A bug in the implementation could wipe out information system-wide. It would be very valuable to have multiple independent implementations of the LCAP protocol. We hope that by keeping the protocol very simple we will encourage other implementations.

Acknowledgments

Grateful thanks for consistent support are due to Michael Keller, the Stanford University Librarian, to Michael Lesk at the NSF for funding the project with grant IIS-9907296, and to Sun Microsystems Laboratories, which has provided both time and funds.

The project was to a large extent inspired by Danny Hillis & Stewart Brand's Millennium Clock project^[clock].

Our thanks also go to the many people who reviewed the design and the paper, especially to Mark Seiden, Bob Sproull, the anonymous Usenix reviewers and Clem Cole; and to Michael Durket (Stanford University ITSS), Tony Smith-Grieco and Demian Harvill (Highwire Press) for help with implementation.

We're very grateful to our long-suffering alpha sites, and to AAAS, who allowed the valuable content of *Science Online* to act as the experimental subject.

References

[**lampport**] "The implementation of reliable distributed multiprocess systems" *Computer Networks* 2, 1978. Butler Lampson provides a useful exegesis "How to build a highly available system using consensus" at <http://www.research.microsoft.com/lampson/58-Consensus/Abstract.html>.

[**highwire**] Highwire Press is at <http://highwire.stanford.edu>. A free example of their work is the British Medical Journal at <http://www.bmj.com>.

[**bit-rot**] Stanford's Conservation Online project maintains a page on this problem at <http://palimpsest.stanford.edu/bytopic/electronic-records/electronic-storage-media/>.

[**format**] Howard Besser maintains a website on this problem at <http://sunsite.berkeley.edu/Longevity/>. The NSF held a March 1999 workshop on it. See <http://cecssrv1.cecs.missouri.edu/NSFWorkshop/execsum.html>.

[**alexandria**] The Encyclopedia Britannica's article on the Library of Alexandria is at http://www.eb.com:180/bol/topic?eu=5704&sctn=1#s_top.

[**lockss**] The LOCKSS project website is at <http://lockss.stanford.edu>.

[**disk-cost**] "The price per megabyte [of disk storage] has declined at 5% per quarter for more than twenty years." Clay Christensen, *The Innovators Dilemma* (1997 Harvard Business School Press).

[**w3mir**] w3mir is supported by Nicolai Langfeldt at <http://www.math.uio.no/~janl/w3mir/>.

[**apache**] The Apache Software Foundation is at <http://www.apache.org/>.

[**srm**] Floyd, S., Jacobson, V., *et al*, "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", IEEE/ACM Transactions on Networking, December 1997, Volume 5, Number 6,

pp. 784-803. See <http://www-nrg.ee.lbl.gov/floyd/srm-paper.html>.

[**enemies**] 1997 was a bad year for libraries - see http://www.eb.com:180/bol/topic?eu=124351&sctn=1#s_top. Education efforts on book mutilation include <http://gort.ucsd.edu/preseduc/bmlmutil.htm>.

[**slp**] SLP is specified by RFC2165 at <http://www.ietf.org/rfc/rfc2165.txt>.

[**ultima**] The FAQs on the Ultima Online reputation system are at <http://update.uo.com/repfaq/>.

[**digest**] SHA-1 is specified by FIPS180-1 at <http://www.itl.nist.gov/fipspubs/fip180-1.htm>.

[**lrp**] The Linux Router Project is at <http://www.linuxrouter.org>.

[**castro**] Castro, M. & Liskov, B. "Practical Byzantine Fault Tolerance", Proc. 3rd Symp. On Operating System Design and Implementation, New Orleans, Feb 1999. http://www.pmg.lcs.mit.edu/~castro/osdi99_html/osdi99.html.

[**archive**] The Internet Archive is at <http://www.archive.org>.

[**nec**] Chen, Y., Edler, J., *et al*, "A Prototype Implementation of Archival Intermemory", Tech. Rept. CEGGSY98, NEC Research Institute, Princeton NJ, Dec. 1998. See <http://www.intermemory.org>.

[**dlr**] The DLI2 initiative is at <http://www.dli2.nsf.gov/>.

[**words**] Phelps, T. A. & Wilensky, R. *Robust Hyperlinks Cost Just Five Words Each* is at <http://HTTP.CS.Berkeley.EDU/~wilensky/robust-hyperlinks.html>.

[**freenet**] Clarke, I., *A Distributed Decentralized Information Storage and Retrieval System* is at <http://freenet.sourceforge.net/Freenet.ps>.

[**napster**] The Napster service is described at <http://www.napster.com>. *Wired* describes the controversial launch of Gnutella at <http://www.wired.com/news/technology/0,1282,34978,00.html>.

[**clock**] The Millennium Clock is a project of the Long Now Foundation at <http://longnow.org/>.

Remember to follow these links before they go 404!

The Globe Distribution Network

A. Bakker, E. Amade, G. Ballintijn

Department of Mathematics and Computer Science

Vrije Universiteit

Amsterdam, The Netherlands

I. Kuz

Delft University of Technology

P. Verkaik, I. van der Wijk, M. van Steen, A.S. Tanenbaum

Vrije Universiteit Amsterdam

arno@cs.vu.nl, <http://www.cs.vu.nl/globe/>

Abstract

The goal of the Globe project is to design and build a middleware platform that facilitates the development of large-scale distributed applications, such as those found on the Internet. To demonstrate the feasibility of our design and to test our ideas, we are currently building a new Internet application: *The Globe Distribution Network*. The Globe Distribution Network, or GDN, is an application for the efficient, worldwide distribution of free software and other free data. The GDN can be seen as an improvement to anonymous FTP and the World Wide Web due to its flexibility and extensive support for replication. This paper describes the design of the GDN. We start by explaining how the replication facilities of the Globe middleware are used to make the GDN efficient, and how these facilities are implemented. Next, we present the architecture of the GDN and discuss how the Domain Name System can be used as a first approach towards a worldwide service for naming software packages and other entities. This is followed by an analysis of the security requirements for the GDN and measures taken to satisfy these requirements. We hope to make Globe and GDN itself available for free under the BSD license by 2001.

1 Introduction

Developing a large Internet application is a difficult task due to the complex nonfunctional aspects that have to be taken into account. A developer has to deal with a potentially very large number of users, high communication delays, security threats, and machine and network failures. The goal of the Globe project is to design and build a middleware platform that facilitates the develop-

ment of worldwide distributed applications by providing extensive support for handling all of these complex nonfunctional aspects [van Steen *et al.*, 1999].

As replication is a powerful technique for dealing with many of these aspects, support for flexible replication plays an important role in the Globe middleware. In Globe, processes communicate by invoking methods on a special kind of distributed object, called a *distributed shared object* (DSO). What makes a distributed shared object special is that we can vary the replication strategy on a per-object basis, allowing the way the object is replicated to be governed completely by object- and application-specific requirements with respect to consistency and nonfunctional aspects, such as security and fault tolerance. Replication and application code are separated, which means that we can reuse replication protocols developed for one distributed shared object to build other DSOs.

The first version of our middleware platform is nearly complete. To demonstrate the feasibility of our ideas and the design of our middleware we are currently building a prototype of a new Internet application using the Globe middleware. This paper describes the design of this application, called the *Globe Distribution Network*. The Globe Distribution Network, or GDN for short, is an application for the efficient, worldwide distribution of data. In the beginning, it will be used for the distribution of publicly redistributable software packages, such as the GNU C compiler, Linux distributions and shareware. We intend, however, to extend this to other types of data, such as free digital music, in the future.

Because it deals with the worldwide distribution of data, the GDN is similar in function to the World Wide Web.

They do differ in one important aspect, however. Although the architecture of the World Wide Web has been shown to be quite scalable, the WWW does suffer from performance problems. We think that these problems are mainly caused by the Web's limited and inflexible support for replication. The GDN therefore needs extensive and flexible replication support. This will be provided by the Globe middleware, via its distributed shared object concept. It is therefore better to compare the GDN with commercial content delivery networks, such as Digital Island's *Footprint* [Digital Island, Ltd., 2000], or wide-area file systems such as AFS [Howard *et al.*, 1998] or CODA [Satyanarayanan *et al.*, 1990].

To avoid any confusion: the purpose of the GDN application is not to replace the Web or become the world's leading distributed file system. It is a research vehicle which should demonstrate the feasibility of our ideas about middleware for large Internet applications. The GDN is a prototype of a system that could one day replace the Web as the Internet application for distributing information, just as the Web has essentially replaced FTP. Nonetheless, it is a serious application that will be running and publicly accessible on the Internet.

Throughout this paper we refer to two versions of the GDN. The first version of the GDN is a limited prototype that will run entirely on machines at our university in June 2000. The version intended to be used by the general public is scheduled to be ready by the end of 2000 and is referred to as the second version of the GDN. The source code of Globe and the Globe Distribution Network will be made available under the BSD license by the end of 2000.

The rest of this paper is organized as follows. Section 2 describes the functionality of the Globe Distribution Network. Section 3 explains how we intend to make the distribution of software packages efficient by using the Globe middleware. After this explanation we present the architecture of the GDN in Section 4. In Section 5 we describe the naming of software packages and our prototype worldwide name service for distributed shared objects. An analysis of the security requirements and the concrete measures we take to meet these requirements are described in Section 6. Section 7 discusses availability of the various Globe and GDN components and gives an overview of their current status. A summary of the paper and our future plans for the GDN can be found in Section 8.

2 The Globe Distribution Network

The Globe Distribution Network is to be a worldwide distributed application for the efficient dissemination of free software packages (e.g. Gimp, *teTeX* and Linux distributions) and other free data. We assume, for the first versions of the application, that a software package has the following basic properties:

1. It consists of one or more files
2. It has a unique name
3. The collection of files or the individual files that are part of the package can be very large

The functionality of the GDN is initially simple: it should be possible to add software packages to the GDN, retrieve copies of software packages, update them and remove packages that are no longer of interest.

We currently divide the user community into three groups: the *GDN users*, the *GDN moderators* and the *GDN administrators*. GDN moderators are allowed to create, update and remove software packages. GDN users are allowed to retrieve packages only. To add a package to the GDN, GDN users must contact a GDN moderator. GDN administrators have complete control over the GDN application and hand out moderator privileges. In the future we intend to introduce a fourth group, the *GDN maintainers*. A GDN maintainer is allowed to manage just the contents of a package. He or she would typically be the person that also maintains the software package (i.e., fixes bugs, etc.). In the first versions we, the Globe team, will play the role of GDN administrators, and together with a number of volunteers act as GDN moderators.

3 Distributing Packages Efficiently

3.1 Flexible Replication

To make software packages available to a worldwide audience they will need to be replicated, for two reasons. First, there are a potentially very large number of people interested in a particular software package and multiple machines are needed to handle such a load. Second, wide-area bandwidth is a scarce resource and with interested people distributed all over the world replicas must be created close to where the clients are (e.g. in each country) to avoid wasting bandwidth. This is, in fact, a trade-off between server capacity (disk space) and bandwidth. Another reason for replicating packages close to clients is the resulting low response time (i.e., a down-

load starts quickly), which is an important usability aspect.

However, replication does not come for free. Each replica of a software package, or, in general, a piece of information, requires a certain amount of disk space and also computing resources while it is being transmitted. Moreover, there is the management aspect: when replicated data is changed, the different replicas have to be made consistent again, and adding or removing replicas to adapt to changes in access patterns is often not fully automated.

For software packages the cost of replication is not a problem. Most countries probably have their own replicas of the complete collection of freely redistributable software packages, distributed over a number of machines throughout the country. The cost of replication does become a problem if we start looking at using the GDN for distributing other types of information. The amount of data that people want to make available to the world is enormous (cf. the Web). Furthermore, the change rates of this data can be much higher.

From this we conclude that for the Globe Distribution Network to be efficient, we should selectively replicate the information we are distributing, based on popularity and update patterns and that the information's *replication scenario* should adapt to changes in its popularity and rate of change. We use the term replication scenario to denote a specification of *how* (using what replication protocol) and *where* (which machines should host replicas) information or objects should be replicated.

We have found evidence to support this conclusion. We analyzed the retrieval and update patterns of our department's Web pages and found that, if we assign a replication scenario to each Web page that reflects that page's individual usage and update patterns, we get significant improvements in a number of areas compared to situations in which a single replication scenario is used for the whole site. In particular, we found that less wide-area network traffic was generated and the response time for the end-user improved [Pierre *et al.*, 1999]. Although this is just one case study, it does suggest that performance problems for large-scale data distribution systems such as the Web and the GDN can be alleviated by introducing more flexible replication capabilities.

We believe that the ability to selectively replicate data is something that is required by all large Internet applications. Therefore, this is an important part of the Globe middleware. Globe is based on the concept of a distributed shared object. The most important aspect of

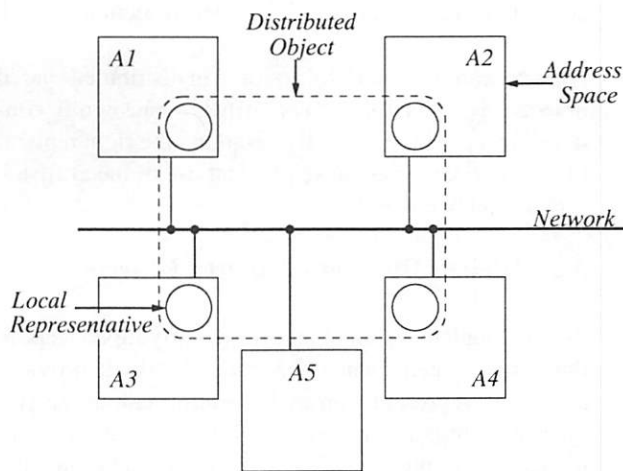
the distributed shared object for the purposes of this paper is that it allows different replication scenarios to be assigned to each object. The distributed shared object concept is discussed in detail in the next section.

All data stored in the GDN is stored in distributed shared objects. For example, every software package is contained in a *package DSO*. By assigning the right replication scenario, we can make efficient use of the available servers and bandwidth.

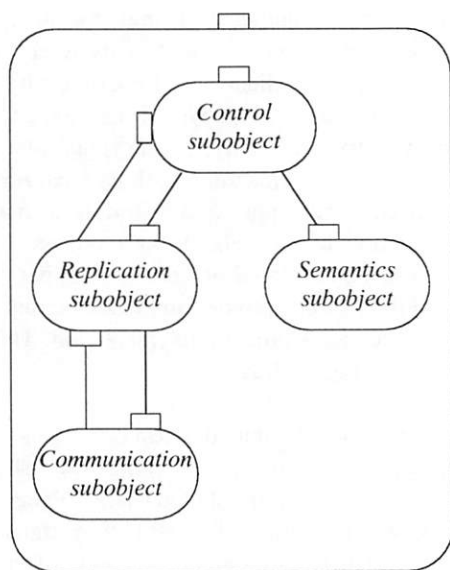
3.2 Globe's Distributed Shared Objects

The distributed shared object is the unifying concept in the Globe system [van Steen *et al.*, 1999]. It provides a uniform representation of both information and services and implementation flexibility by decoupling interface and implementation. The fundamental idea behind the design of the distributed shared object is that it is *physically distributed*. Most current middleware, such as CORBA [Object Management Group, 1999] and DCOM [Eddon and Eddon, 1998], view a distributed object as an object running on a single machine, possibly with copies on other machines. This object (group) is presented to remote clients as a local object by means of proxies. In contrast, we view a distributed shared object as a distributed entity, a conceptual object distributed over multiple machines with its *local representatives* (proxies and replicas) cooperating to make the object's functionality available to local clients. In other words, a distributed shared object is a wrapper encompassing all the object's proxies and replicas, rather than a remotely accessible object implementation. This view is illustrated in Figure 1(a).

Our view of what a distributed object is gives us flexibility with respect to replication, caching and distribution of the object's state. A distributed shared object encapsulates its own replication and distribution strategy. The local representatives of an object take care of the replication and distribution of the DSO's state and all necessary communication. Only minimal (protocol independent) support is required from the run-time system. This means that the way the state of the object is replicated can now be governed completely by object- and application-specific requirements with respect to consistency and nonfunctional aspects, such as security, and is under no restriction from the supporting middleware platform. However, we do not leave everything to the application programmer. The structure of local representatives, described below, separates replication and communication code. This means that a programmer can write his or her own replication protocol based on existing communication protocols. Furthermore, we pro-



(a)



(b)

Figure 1: (a) A distributed shared object (DSO) distributed over four address spaces (A1-A4). In each address space the DSO is represented by a local representative. Address space A5 does not currently contribute to the distributed shared object. (b) A local representative is composed of a number of subobjects. The exact composition depends on the role the local representative plays in the distributed shared object.

vide the application programmer with implementations of frequently used replication protocols.

3.3 Implementation of the Globe Object Model

In this section we describe how this object model can actually be implemented. Logically, a DSO consists of multiple local representatives. A local representative resides in a single address space and communicates with local representatives in other address spaces. Each local representative is composed of several subobjects as shown in Figure 1(b). A typical composition consists of the following four subobjects.

Semantics subobject: This is a local object that implements (part of) the actual semantics of the distributed object. As such, it encapsulates the functionality of the distributed object. The semantics subobject consists of user-defined primitive objects written in programming languages such as Java, C, or C++. These primitive objects can be developed independent of any distribution or replication issues. In the case of a package DSO this subobject would implement all the DSO's methods, such as methods for adding files to a package, for listing the files currently in a package and for retrieving the contents of a file.

Communication subobject: This is generally a system-provided subobject (i.e., taken from a library). It is responsible for handling communication between parts of the distributed object that reside in different address spaces, usually on different machines. Depending on what is needed from the other components, a communication subobject may offer primitives for point-to-point communication, multicast facilities, or both.

Replication subobject: The global state of the distributed object is made up of the state of the semantics subobjects in its local representatives. A DSO may have semantics subobjects in multiple local representatives for reasons of fault tolerance or performance. In particular, the replication subobject is responsible for keeping the state of these replicas consistent according to some (per-object) coherence strategy. Different distributed objects may have different replication subobjects, using different replication algorithms. For example, one object may actively replicate all the state at all the local representatives while another may use lazy replication. An important observation is that the replication subobject has standard interfaces.

Control subobject: The control subobject takes care of invocations from client processes, and controls the interaction between the semantics subobject and the replica-

tion subobject. This subobject is needed to bridge the gap between the user-defined interfaces of the semantics subobject, and the standard interfaces of the replication subobject.

A key role, of course, is reserved for the replication subobject. Replication (and communication) subobjects are unaware of the methods and state of the semantics subobject. Instead, both the replication subobject and the communication subobject operate only on opaque invocation messages in which method identifiers and parameters have been encoded. This independence allows us to define standard interfaces for all replication and communication subobjects. This approach is comparable to techniques applied in reflective object-oriented programming [Kiczales *et al.*, 1991].

3.4 Binding to a Distributed Shared Object

To access a distributed shared object (i.e., to invoke its methods), a client first needs to install a local representative of the object in its address space. The process of installing a local representative in an address space is called *binding*. Before we explain binding, however, we first have to describe how naming is done in the Globe middleware.

Each DSO in Globe is identified by a worldwide unique object identifier (OID). This object identifier, or *object handle*, never changes during the lifetime of the object and, most importantly, is location independent. The actual locations of the DSO, that is, *where* (network address, port number) its local representatives are located, and *how* (which replication and communication protocol) they can be contacted is maintained by a special service, the *Globe Location Service* (GLS) [van Steen *et al.*, 1998]. Typically only local representatives acting as replicas are registered in the GLS. The information that identifies the location of a local representative and how to talk to it is called a *contact address*. The set of contact addresses stored in the GLS for a specific DSO describes that object's replication scenario.

Object identifiers are long strings of bits and thus unusable for humans. We therefore have an additional name service which maps symbolic names to object identifiers. This results in two-level naming scheme: symbolic object names are mapped to object identifiers by the *Globe Name Service* (GNS) which are, in turn, mapped to one or more contact addresses for the object by the *Globe Location Service*. The inner workings of the *Globe Location Service* are described in the next section. Our prototype of the *Globe Name Service* is discussed in Section 5.

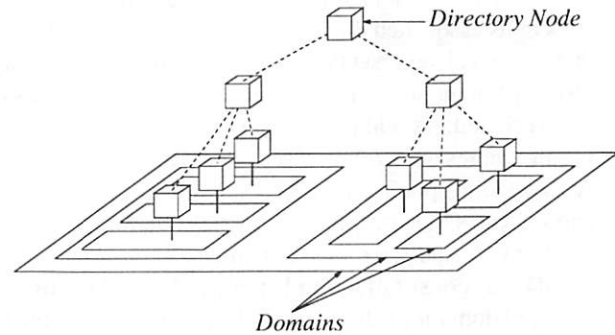


Figure 2: The *Globe Location Service* divides the Internet into a hierarchy of domains, represented by rectangles in the figure. Associated with each domain is a directory node, represented by a little box.

Binding to a DSO now works as follows. For brevity we assume the client has already acquired an object identifier of the DSO whose methods it wants to invoke. The client calls a special function in the run-time system, named *bind*, and passes it the object identifier. The run-time system takes the OID and asks the *Globe Location Service* to map this OID to one or more contact addresses. In general, the returned contact addresses will identify the nearest replica of the DSO. Using the information in the contact addresses, the local run-time system then creates a new local representative in the client's address space and integrates this new representative into the DSO. This involves loading the implementation of the local representative (i.e., the appropriate set of subobjects) from a nearby implementation repository in a way similar to remote class loading in Java.

3.5 The Globe Location Service

To efficiently map object identifiers to contact addresses on a worldwide scale, we organize the Internet into a hierarchy of *domains*. The domains at the bottom of the hierarchy represent moderately-sized networks, such as a university's campus network or the office network of a corporation's branch in a certain city. The next level in the hierarchy is formed by combining these leaf domains into larger domains (e.g. representing the city's MAN). This procedure is applied recursively until the root domain which encompasses the whole Internet. Note that domains in this hierarchy do not necessarily correspond to DNS domains.

With each domain in the hierarchy we associate a *directory node*, as shown in Figure 2. Each directory node

keeps track of the locations of the distributed shared objects in its associated domain, as follows. For each DSO that has local representatives in the node's domain, a directory node stores either the actual contact address (network address and protocol information for contacting the representative) or a set of *forwarding pointers*. A forwarding pointer points to a child directory node and indicates that a contact address can be found somewhere in the subtree rooted at that child node. Because a DSO may consist of multiple replicas located in different child domains, a directory node may store more than one forwarding pointer per DSO. Normally, the contact addresses are stored in the leaf directory nodes. However, storing the addresses at intermediate nodes may, in the case of highly mobile objects, leads to considerably more efficient look-up operations, as we explained in [van Steen *et al.*, 1998]. This design has some (apparently) radical consequences. For each DSO on the Internet, there is a tree of forwarding pointers from the root node to the directory nodes that contain the actual contact addresses. Before we explain that this, in fact, does not create a single point of failure or bottleneck, we first look at how object identifiers are resolved.

During binding, the (run-time system of a) client sends a look-up request to the directory node of the leaf domain the client is located in. The leaf node checks if it has a contact address for that DSO in its tables (i.e., it checks if the DSO has a representative in this (leaf) domain). If not, it forwards the request to its parent node, which, in turn, checks its tables. This process is repeated until either a contact address for the object is found or a forwarding pointer is discovered. In the latter case, the look-up operation continues down into the subtree pointed to by the forwarding pointer and follows the tree of forwarding pointers to the node in that subtree that stores the actual contact address. If multiple forwarding pointers are found, one is chosen at random.

The advantage of this design is, that if a distributed shared object has a representative near to the client, the Globe Location Service will find that representative using only "local" communication. In other words, the cost of a look up increases proportional to the distance between client and nearest representative.

The apparent problem with this design is that the root node, or in general, the higher-level nodes in the hierarchy have to store a lot of forwarding pointers and handle a lot of requests (if representatives of the DSO are not located near their prospective clients). Our solution to this problem is to partition a directory node into one or more *directory subnodes*. Each subnode is made responsible for a specific part of the object-

identifier space via a special hashing technique and can run on a separate machine. For further details we refer to [Ballintijn and van Steen, 1999a].

4 The GDN Architecture

Having explained the distributed shared object concept, we can now describe the basic architecture of the GDN application. The core of the application is a set of *Globe Object Servers* (GOSs), running on machines all over the world. A Globe Object Server is an application-independent daemon for hosting replicas of any kind of distributed shared object. Globe Object Servers allow replicas to save their state during a reboot and reconstruct themselves afterwards. The set of GOSs hosts the replicas of the DSOs containing the software packages.

To access the contents of a package DSO, a user would normally have to start up a tool that binds to the distributed shared object and allows the user to invoke methods on that package DSO. The disadvantage of this approach is that users have to run a dedicated client to access the GDN. We want to make the threshold for users to access the GDN as low as possible, and have therefore decided to make the GDN accessible through standard Web browsers. Furthermore, being able to access the GDN via a Web browser allows use to easily integrate it with the World Wide Web.

As such the GDN also consists of a number of modified HTTPDs running on machines all around the world. In our first versions they will be colocated with the Globe Object Servers. These modified, or *GDN-enabled* HTTPDs work as follows. We use URLs that have embedded in them the name of a package DSO. The GDN-HTTPD extracts this object name and binds to the DSO. The HTTPD then invokes the appropriate method(s) on the package DSO's newly created local representative. For example, it could call `listContents()` to obtain the list of files contained in the package, which is subsequently reformatted into HTML and sent back to the requesting browser. If the URL designates a particular file in the package, the HTTPD calls the `getFileContents()` method and sends back the returned content. The local representative that is installed in the GDN-HTTPD during binding may act as a replica for the DSO, in which case downloading a software package is fast.

Users communicate with only one GDN-HTTPD, in particular, with the one nearest to them. This HTTPD is the user's access point to the GDN. We currently require users to manually select this HTTPD, using a list published on a central web site. Once connected to the

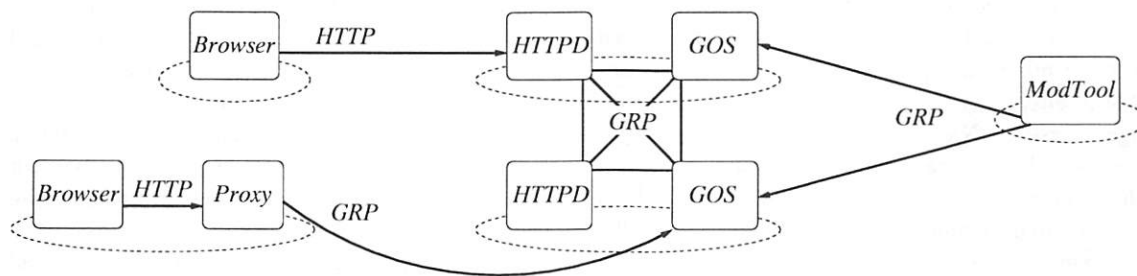


Figure 3: The architecture of the GDN application. Ovals represent sites, rounded boxes represent programs running at those sites, arrows and thick lines represent communication. GRP stands for Globe Replication Protocol. ModTool is the moderator tool. Omitted from this figure are the programs belonging to the Globe Location and Name Services.

GDN, however, the storage location of software packages becomes transparent. The GDN will transparently find the nearest replicas using the Globe Location Service.

Using standard Web browsers is fine, but we would also like people to use the GDN directly. To this extent we allow people to run GDN-HTTPDs on their own machines. We refer to these GDN-HTTPDs as *GDN-enabled proxy servers*, or GDN-proxy servers for short. The last element in the architecture are the moderator tools. A GDN moderator (see Section 2) can add, update and delete package DSOs from the GDN, using a special tool. Figure 3 shows the complete architecture of the GDN.

5 Naming Packages

Most software packages have unique names and people should, of course, be able to retrieve and update packages from the GDN using that name as a key. In addition, we would like the GDN to support some form of attribute-based search, such that people can look for a software package with some specific functionality.

We introduce a hierarchical name space in which the first part of the name gives some information about what a software package does. For example, the *Gimp* graphics package would be named something like */apps/graphics/Gimp* to indicate that it is a package for manipulating graphics. A package is allowed to have more than one name so we can have multiple classifications. Having a hierarchical name space also allows us to name DSOs other than packages in a separate name subspace in the future. The exact structure of the name space (*/apps*, */os*, */middleware*, ...) is outside the scope

of this paper.

As described above, the assignment of human-readable names to distributed shared objects is handled by the Globe Name Service (GNS). The GNS found in the current Globe middleware is a prototype version based on the Domain Name System [Mockapetris, 1987]. The reason for using DNS is that we wanted to build a reasonable name service in a short period of time, so we took an existing system that was suitable for our purposes.

DNS maps symbolic names to other types of data and scales to large numbers of users. DNS works under the assumption that the mapping of names to addresses does not change very frequently. This allows the DNS to cache entries at client-side resolvers and to replicate parts of the database on multiple machines. Combined with distributing the mapping of names to addresses across hosts this results in a scalable system. We can make that same assumption: we expect our name-to-object-identifier mappings to be stable, because of the two-level naming scheme of Globe.

The DNS-based version of the Globe Name Service works as follows. Globe object names have a one-to-one mapping to valid DNS names. These DNS names point to a TXT DNS Resource Record that contains the encoded object identifier for the DSO. To map a Globe object name, say */nl/vu/cs/globe/somePackage*, to a Globe object identifier, the object name is first translated to a DNS name, in this case *somePackage.globe.cs.vu.nl*. This DNS name is then resolved using the normal DNS name resolution mechanism and returns a TXT record from which the object identifier is extracted.

An advantage of this approach is that there is a global

name space for objects (the DNS name space) and anyone in control of a DNS domain can create their own subspace in this name space which is immediately accessible to anyone in the world. There are also disadvantages. Firstly, DNS places restrictions on name syntax (i.e., which characters can be used in a name and how long the individual parts of a name can be) which have been lifted in modern name systems. Secondly, DNS domain names are always part of object names, which is not always desirable. Thirdly, the current DNS is insecure because it is vulnerable to spoofing attacks [Vixie, 1995]. We come back to this issue in Section 6.

For the Globe Distribution Network, we intend to work around the second disadvantage. We do not want users to see the DNS domain, we want them to be able to use names such as /apps/graphics/Gimp. To achieve this we use only a single DNS leaf domain to register the names of package DSOs. This means that we can omit the DNS domain name part from the package DSO's name, given that we also modify the GDN software to always prefix this DNS domain name before it is passed to the Globe Name Service. We refer to this DNS leaf domain as the *GDN Zone*.

We expect that this will not cause problems for the first two versions of the GDN. For these versions, we control the addition and naming of package DSOs to the GDN and we can distribute the load by creating multiple authoritative name servers. The number of updates to our zone can be kept low by batching them. For later versions we hope to replace the DNS-based prototype with a GNS based on distributed shared objects [Ballintijn and van Steen, 1999b].

6 Security

6.1 Security Requirements

An important aspect of any new Internet application is security. We discuss security of the Globe Distribution Network in three parts. First, we identify the security requirements for the GDN. We start by identifying the requirements at a high level of abstraction and then translate them into more specific requirements. Second, we describe the security situation, that is, the assumptions we make about the machines on which the GDN will run and their network environment. Finally, we describe the concrete measures we take to satisfy the identified security requirements given the environment in which the GDN will operate.

An important security requirement is that the GDN ap-

plication is protected against unauthorized use. It should not be possible to use the GDN for the unlawful distribution of commercial software, copyrighted music and such. In the beginning, the GDN will be used primarily for distributing software packages which results in two additional security requirements: attackers should not be able to violate the integrity of the software being distributed and users of the GDN should be assured of the origin of the software. Another requirement is availability. Like the Web and FTP, the GDN application should be highly available and measures should be taken to fend off attacks intended to stop the application from operating. Other factors threatening availability are host and network failures. How these failures are handled in the GDN, and more general, in the Globe middleware is still an open research issue, but replication is, of course, one technique. These high-level requirements can be translated into more specific requirements, as follows.

Unauthorized Use

Only a GDN moderator should be able to add packages, names, etc. to the GDN. This (a) prevents people from filling the GDN with junk packages (i.e., denial of service through resource allocation) and (b) it prevents the GDN from being used for the illegal distribution of copyrighted data. We can further split this up into a number of subrequirements.

Adding and Removing Packages Only a moderator should be able to add and removing packages to the GDN. Adding a package DSO consists of a number of steps, which are executed by the moderator tool (see Section 4). The creation of a new package DSO starts with the definition, by the moderator, of the package's replication scenario. Recall that the replication scenario of a DSO describes how (using what replication protocol) and where (which machine(s) should host replicas) a DSO should be replicated. The moderator tool will present the moderator with the choice of available replication protocols and the set of available Globe Object Servers.

When the replication scenario has been defined, the moderator tool starts sending commands to the chosen Globe Object Servers. It starts by sending a "create first replica" command to one (randomly chosen) GOS in the scenario. This Globe Object Server constructs a local representative for that DSO in its address space, and registers a contact address for this local representative in the Globe Location Service. As part of the registration, an object identifier is allocated for the DSO by the GLS. This object identifier is returned to the moderator tool. The other GOSs are then sent "bind to DSO <OID>,"

create replica” commands. The replicas they create are also registered with the GLS.

The final step in creating a package DSO is registering a name for it in the Globe Name Service. To this extent the moderator tool calls a library routine which communicates with the GNS. In particular, this library routine contacts the so-called *GNS Naming Authority* for the GDN Zone. This is the daemon that sends DNS UPDATE messages [Vixie *et al.*, 1997] to the name servers responsible for the GDN Zone, in response to add and remove requests from clients.

Looking at this procedure, we can derive three security subrequirements:

1. A Globe Object Server should accept only commands sent by a GDN moderator.
2. The Globe Location Service should accept only object registrations (and deregistrations) from Globe Object Servers which are officially part of the GDN.
3. A GDN Naming Authority should accept only updates from moderator tools operated by official GDN moderators.

Modifying Packages Without loss of generality, we can say that to ensure the integrity of the data inside a package DSO, Globe Object Servers and GDN-enabled HTTPDs (i.e., the processes potentially hosting replicas of the DSO) should not accept state-modifying method invocations and state update messages from unauthorized senders. Authorized senders are: (1) a moderator tool operated by an official GDN moderator and (2) Globe Object Servers that are part of the GDN (e.g. a Globe Object Server acting as master replica in a master/slave replication protocol). This is, of course, not a sufficient condition. We should also protect servers from direct tampering through break ins on the machines they are hosted on.

Availability

People should not be able to crash our critical servers, nor render them inoperable using bogus protocol messages. The critical servers in the GDN are:

- Location Service directory nodes and auxiliary GLS daemons
- Object Servers
- GDN-enabled HTTPDs

- DNS servers and auxiliary daemons used by the DNS-based GNS

6.2 Operating Environment

We assume the following security situation. The different parts of the GDN application run on machines distributed all over the Internet, however, the critical parts of the application, such as the Globe Object Servers, the Location Service’s nodes and moderator tools run only on secure machines. By secure we mean that only authorized personnel can install software on them, log in, etc. We call these machines the *GDN hosts*. For the first versions of the GDN we assume that the networks connecting the GDN hosts cannot be tapped by attackers. These networks are not, however, firewalled, so anyone on the Internet can send network packets to these hosts.

We consider the parts of the application that are running on users’ machines to be insecure. These are the GDN-enabled proxy servers and the users’ browsers (see Section 4). Furthermore, we assume the connections between the GDN hosts and the users’ machines are not secure. The last aspect of the security situation is that the source code of both the GDN application and Globe are publicly available, which makes staging an attack simpler.

6.3 Security Measures

As the security framework for Globe is still under development and will not be incorporated into Globe before the end of 2000, we will not be able to use it in the first versions of the GDN. Instead we will develop a more limited, GDN-specific security model for these versions.

Because the first (June 2000) version will run in a controlled environment we will not actually implement any security measures until the second version. To secure this version we replace all communication between GDN parties by integrity-protected and authenticated communication. In particular, all TCP connections between GDN parties are replaced by connections secured via the TLS protocol [Dierks and Allen, 1999] and its predecessor, the Secure Sockets Layer (SSL) [Freier *et al.*, 1996]).

TLS offers one-way or two-way authenticated communication channels which are encrypted and protected against content modification. The idea is that GDN hosts use two-way authenticated channels for internal communication, and server-side authentication for all communication with software running on users’ machines (i.e., browsers or GDN-proxy servers). This situation is illus-

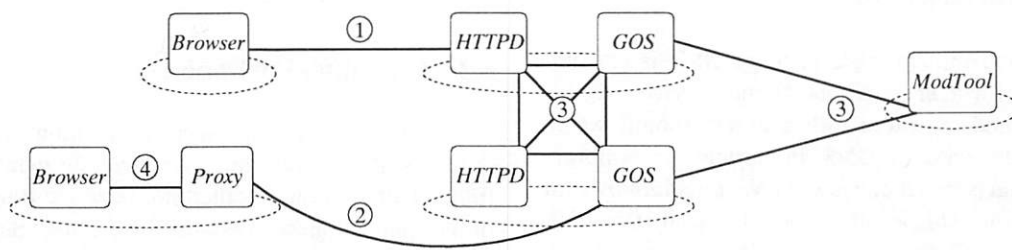


Figure 4: We secure the GDN using a transport-layer security library. Communication between GDN hosts, marked (3) in the figure, is fully authenticated. Communication between GDN hosts and browsers running on users' machines (marked (1) in the figure) is authenticated one way: the GDN host authenticates itself to the users' machines. This is also the case for communication between GDN hosts and GDN-enabled proxy servers on users' machines ((2) in the figure). If desired, a user could configure a GDN-enabled proxy server to also authenticate itself to the local browsers ((4) in the figure), but we consider this a local administrative matter.

trated in Figure 4.

By making sure that sensitive requests, such as state-modifying method invocations, are executed only when sent from authenticated hosts (i.e., GDN hosts) attackers are not able to compromise the integrity of the data contained in the GDN. This also protects software packages downloaded via browsers from malicious modifications.

For the GDN to work we will, however, still have to accept network traffic from unauthenticated hosts (in particular, from users' machines). This means that attackers are potentially able to crash the GDN, that is, compromise availability by sending malformed packets which cause the GDN to crash. We intend to counter these type of attacks by good programming, avoiding buffer flows, etc. We will not take extra measures against denial of service through flooding.

A disadvantage of this scheme is, of course, that we are paying for something we do not need: confidentiality. TLS and SSL provide confidentiality as well as authentication and integrity protection. We are interested only in the latter two. If performance is affected too negatively by the superfluous encryption and decryption we will have to rethink our security scheme.

This solution seems to be quite feasible in practical terms. To implement this security scheme we need to rewrite our communication layers to use TLS or SSL. This should not require too much effort since we have cleanly separated communication from functional layers in all our software (e.g. see Section 3.3) and TLS/SSL builds on the BSD socket interface. Availability of a TLS/SSL library for Java, the language in which all our

software is written is a potential issue. Fortunately, Sun has recently published the Java Secure Sockets Extension (JSSE) [Sun Microsystems, Inc., 1999] that implements TLS and SSL.

The JSSE package can be legally exported from the US. The only potential problem is usage restrictions in the countries where the GDN hosts are located. At this point in time we have no knowledge about what machines will be available to us, therefore we cannot assess how serious a problem this is.

The one case where the TLS scheme cannot be used is the Globe Location Service. For efficiency reasons this is based on UDP. We have yet to determine if it is acceptable to temporarily replace it with TCP, or that we should implement a specific security scheme for the GLS.

Another special case is the DNS-based Globe Name Service. The problem is that this TLS/SSL scheme can be used to secure only the connection between the GDN Naming Authority, that is, the daemon which sends DNS UPDATE messages and the moderator tools that request these changes. We cannot protect the DNS itself using this method, for obvious reasons.

The effects of DNS spoofing on the GNS, and the GDN in general, are limited, however. Basically, attackers can only prevent resolution of object names to object identifiers or cause an object name to resolve to an invalid object identifier or to one belonging to another object. Denial of service attacks on other parts of the GDN can be prevented if we use IP-addresses instead of DNS names for internal GDN configuration. Our use of TLS and BIND's TSIG security feature (the GNS is build on

BIND8 [Internet Software Consortium, 2000]) will prevent abuse or modification of the GDN's contents.

7 Availability and Current Status

The source code of the GDN and Globe will be made available through the Globe WWW site located at

<http://www.cs.vu.nl/globe/>

The current (March 2000) status of the GDN is as follows. We are writing the control and semantics object for the package DSOs and will then start working on the moderator tool and the GDN-HTTPD. For the latter we can build heavily on an earlier prototype. The current status of the Globe middleware can be described best by looking at what steps are necessary to create a new kind of distributed shared object and to get an application using an instance of that kind of DSO up and running.

The application programmer starts by defining the interfaces of the DSO¹ in Globe's interface definition language (IDL). Using our IDL compiler these interfaces are translated into Java. Using these translated definitions the application programmer writes two subobjects: the semantics subobject that implements the actual functionality of this kind of DSO and the control subobject (see Section 3.3). Control subobjects should be generated automatically in the future.

These implementations are copied to all machines that need to run local representatives of DSOs of this kind and placed in the local implementation repository (currently a directory in the local file system). The last step in writing a Globe application is to write the clients that use the DSO. The Globe part of these clients is easy to implement. The programmer should initialize the run-time system and ask it to bind to a given object identifier, after which the client can access the DSO via its local representative.

To actually run the application the application programmer first has to start and configure the name and location services. Our current Java implementation of the Globe Location Service supports the basic look-up, insert and delete operations and, in addition, persistent storage of the state of a directory node (location information and forwarding pointers). We are in the process of adding a simple crash recovery mechanism to this implementation. The source code of the Java-based GLS cannot be released due to contractual agreements until January

¹Globe uses a model in which an object can have multiple interfaces, as in Microsoft's COM.

2001. Releases of Globe prior to that time will contain the GLS in byte-code form.

The DNS-based prototype of the Globe Name Service is implemented on top of BIND8 [Internet Software Consortium, 2000]. It is fully functional, meaning that a user can add, resolve, change, and delete object names and directories via routines in the Globe run-time system. These routines communicate with the GNS Naming Authorities and through it, with the name servers for the domain the updates are made in.

Once the application programmer has the services up and running, he or she starts up a number of Globe Object Servers and instructs them to create an instance of the DSO with a particular replication scenario. The application is now ready to be used. There are currently two replication protocols an application programmer can choose from: client/(single) server and master/slave. The Globe Object Server and supporting tools are currently being implemented and should be part of the first public Globe release.

8 Conclusions

The goal of the Globe project is to design and build a middleware platform that facilitates the development of Internet applications. These applications are characterized by the complex nonfunctional aspects their programmers have to take into account: potentially huge numbers of users, high communication delays, host and network failures. An important technique for dealing with these aspects is replication of data and functionality, making it an important topic in the Globe middleware. We think that Globe's distributed shared object concept, combined with a worldwide location service for tracking the whereabouts of these distributed objects can offer the flexibility with respect to replication that Internet applications require.

The first version of our middleware platform is nearly complete. To demonstrate the validity of our design and ideas we are building a prototype application. This application, the Globe Distribution Network, or GDN, is a distributed application for the efficient, world-wide, distribution of data. This data initially consists of publicly redistributable software packages. Although comparable in function to the World Wide Web, the GDN has one important advantage, namely the builtin support for replication that it inherits from the Globe middleware.

Current GDN functionality is simple: software packages

can be added, retrieved and removed from the Network. Two possible functional additions we are considering are a more powerful mechanism for attribute-based search and version-management facilities. In the nonfunctional arena, fault tolerance is a topic that needs to be addressed. The naming of software packages is currently done by a prototype object name service that builds on the Domain Name System. Furthermore, we currently use the Transport Layer Security (TLS) protocol to satisfy the GDN's security requirements. We intend to replace these two parts with properly designed name and security services in the future.

The GDN will be usable as an Internet application in December 2000. The source code of the GDN and of the Globe middleware will be released under the BSD license during the course of 2000.

Acknowledgments

The Globe team would like to thank Stichting NLnet and Océ for their support in the development of Globe and the Globe Distribution Network.

References

- [Ballintijn and van Steen, 1999a] G. Ballintijn and M. van Steen. Exploiting Location Awareness for Scalable Location-Independent Object IDs. In *Proceedings Fifth Annual ASCI Conference*, pages 321–328, Heijen, The Netherlands, 1999. Advanced School for Computing and Imaging.
- [Ballintijn and van Steen, 1999b] G. Ballintijn and M. van Steen. Scalable Naming in Global Middleware. Technical Report IR-464, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, October 1999.
- [Dierks and Allen, 1999] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, January 1999.
- [Digital Island, Ltd., 2000] Digital Island, Ltd. Footprint. <http://www.digisle.net/services/cd/footprint.shtml>, March 2000.
- [Eddon and Eddon, 1998] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.
- [Freier et al., 1996] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0, November 1996. Netscape Communications, Inc., Mountain View, CA.
- [Howard et al., 1998] J.H. Howard, M.L. Kazar, S.G. Meneses, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1998.
- [Internet Software Consortium, 2000] Internet Software Consortium. BIND. <http://www.isc.org/products/BIND/>, March 2000.
- [Kiczales et al., 1991] G. Kiczales, J. Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
- [Mockapetris, 1987] P. Mockapetris. RFC 1034: Domain Names – Concepts and Facilities, November 1987.
- [Object Management Group, 1999] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.3.1. OMG Document formal/99-10-07, Object Management Group, Framingham, MA, October 1999.
- [Pierre et al., 1999] G. Pierre, I. Kuz, M. van Steen, and A.S. Tanenbaum. Differentiated Strategies for Replicating Web Documents. Technical Report IR-467, Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, November 1999.
- [Satyanarayanan et al., 1990] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.
- [Sun Microsystems, Inc., 1999] Sun Microsystems, Inc. Java Secure Socket Extension. <http://java.sun.com/products/jsse/>, August 1999.
- [van Steen et al., 1998] M. van Steen, F.J. Hauck, and A.S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, pages 104–109, January 1998.
- [van Steen et al., 1999] M. van Steen, P. Homburg, and A.S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, pages 70–78, January 1999.
- [Vixie et al., 1997] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE), April 1997.
- [Vixie, 1995] Paul A. Vixie. DNS and BIND Security Issues. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, Salt Lake City, Utah, June 1995.

Open Information Pools

Johan Pouwelse

Delft University of Technology, The Netherlands

`j.a.pouwelse@its.tudelft.nl`

Abstract

On the WWW it is not possible to supplement existing web pages of other people with new information or a link to that information, because the WWW does not have a standard method for write access. With write access, information can be added in the right context, which eases searching. We therefore define Open Information Pools: a collection of WWW based databases with public write access. By using databases we add structure to the information. Each database deals with a specific topic. We developed an architecture to support Open Information Pools. Important elements in the architecture are the rating and moderation tools. With these tools the user group is able to maintain and update the database and also to prevent errors and abuse.

We conducted measurements on operational rating and moderation tools to show the validity of our idea. The study of Slashdot.org's rating and moderation tools shows that insightful information is recognised after only 37 minutes. We implemented a prototype of a true Open Information Pool containing music information. This database contains biographies, audio CD descriptions, audio CD cover pictures, lyrics of the songs with timing information, and MIDI files. We developed several tools to create, insert and search this database.

1 Introduction

Several people have dreamed of building a system that could unlock the knowledge of humanity. The MEMEX system [1], Xanadu [17], and the World Wide Web (WWW) are steps to realise that dream. Inspired by these ideas we propose a system that is one step further to the realisation of that dream.

The WWW contains vast amounts of information, without any structure. The lack of structure within WWW is both its strength and its weakness. Finding information in the vastness of the WWW is a serious problem. It also lacks some features for access and addition of information.

First, the WWW lacks the possibility of writing information in context. In the classic paper of Vannevar Bush [1] the addition of information to the collection was identified as a mandatory feature. However, WWW pages on the Internet do not have write access. Only a limited and non-standard form of write access can be provided through the use of CGI scripts. A contribution of information can *not* be made to the point at which it appears on the WWW. The inability of supplementing already present information with a bi-directional link inhibits the true accumulation of information. Information cannot be placed on the WWW within the context to which it belongs, instead it is tied to a "site-name". As a consequence, search engines have to restore the context by giving the translation of keywords to relevant locations. Second, the WWW has no standard rating and moderation systems. Between the outdated, irrelevant, and incorrect content on the Internet lie the true gems of information. Without a built-in mechanism to rate and moderate information it is impossible to make a distinction. We need to learn from the mechanisms that already exist for a long time in the "paper world". Third, there is no direct support for replication and synchronisation within WWW. A system that contains all knowledge of humanity needs to be distributed across the globe. Without any distribution the central points of the system would break down. Mirroring of sites and proxies are only added solutions to replicate the WWW. The WWW is not a fully distributed system, it is a collection of interconnected, yet independent HTTP daemons.

The WWW represents a significant advancement for the dissemination of knowledge, it can be viewed as

a 15-billion-word encyclopedia [3, 12]. The significant growth in size, access, and reach of the WWW make it vital that new systems are enhancements of the WWW instead of competitive replacements. Addition of the above features to the WWW is difficult. Write access is a nightmare for security. Rating and moderation is difficult because ratings are highly individual and depend on a personal point of view. Distribution is an ambitious step compared to the current WWW practice.

The basis of our *open information pools* concept is our belief in openness. Information does not appear, but must be provided by someone. The obstacles for adding information must be as small as possible. The organisation of this information must be as open as possible: everybody must be able to contribute or copy content. With this openness we are able to solve the three problems of the WWW.

This paper applies the ideas of Open Source software to collections of structured information. The Open Source idea is that software must be distributed as source code, enabling everybody to improve and extend this software [9]. Due to a special clause in the GPL copyright notice, often used by much of the Open Source Software community, software under a GPL copyright must remain in the public domain. The philosophy of keeping something open has been applied to other fields besides software as well. For example, an initiative has been started to keep web content open [25], and another to publish hardware designs [29].

This paper proposes to create open information pools, with the underlying philosophy that *information needs to be in the public domain*. We define an open information pool as “*a structured collection of text, pictures, movies, sound, and other data, which may be freely copied and to which the whole Internet community can add information*”. The structured pools of information are implemented as relational databases. Each database contains information about a specific topic in several tables. Everybody can contribute to such a database and also copy the content, provided that added information remains public. Rating and moderation systems are used to keep the content “clean”. The open form makes it possible to create information pools that are impossible, or not cost effective, to create and maintain by a firm. When a firm owns and maintains a database, the database can disappear when for example the firm goes out of business. Public information has a smaller risk of disappearance.

In [9] Raymond discusses the gift culture of Open Source. With our open information pools we will try to show that the same gift culture can also be cultivated in the information domain, besides the software domain. Within Open Source it has been shown that people are willing to freely share their work without any direct payment. The quality of this shared work can even surpass the quality of the work produced within firms that do not use this gift culture. On the Internet people are willing to share information freely, for example, within Usenet. People can be motivated to share information for free for the satisfaction of a good reputation amongst their peers, the knowledge that a good reputation can also pay-off in the real world, or pure altruism.

By using the Internet as the basis of the information pools, world wide access is guaranteed. Adding information to Internet-based databases is already possible [20, 21, 23, 24, 25, 26, 27, 28]. The majority of these databases do not contain scientific, medical, historical, or cultural knowledge but “popular information”, with a wide audience. Unfortunately, several of these databases have been taken out of the public domain for commercial reasons. Users cannot copy the entire content of those non-public domain databases. We believe that this protective commercialism is a dangerous development that threatens the free flow of information. The Internet movie database [28] (about 200,000 entries), and the audio CD database [21] (about 500,000 entries) were open, but are now closed. The audio CD database is popular, with more than 600 contributions of information *daily*. However, the quality of the content is low; the number of duplicate records is measured to be as high as 12 in some cases. Slashdot.org, in contrast, is a good example of how an open information pool should operate. Slashdot.org is a web site with an open news and discussion forum, it has public write access, is based on a relation database, uses an advanced moderation system, and has a large user base. The Open Directory Project [26] is also a good example, they have created a comprehensive directory of over a million web pages, by relying on a “vast army” of volunteer editors. The directory can be freely copied, and is used by Netscape, Lycos, HotBot, and others.

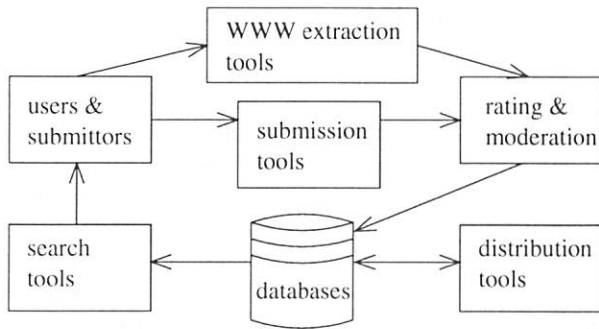


Figure 1: Open Information Pools architecture

2 System overview

We designed an architecture that supports open information pools and adds to the WWW the three missing features discussed in Section 1. The architecture consists of several components, see Figure 1. Current Internet based writable databases are tightly bound to the database content. In our architecture, every component is independent of the information pools content. With our generic architecture, the number of these open, Internet based information pools could grow and software could be re-used.

2.1 Users and submitters

The submitters are the key to the whole open pool concept. The database must be of sufficient interest in order for them to freely submit the information they have. People that have bought a described item, are fan of the subject, or do research on it are especially motivated to submit. Computer literacy also plays a role, computer related database entries such as security vulnerabilities, 2nd hand computer adds, FAQs, manuals, and computer science encyclopedia could be popular. When information is submitted it is also important to register the submitter as a form of recognition for his efforts. People with a high number of submissions build a certain reputation with this work. The motivations behind submission are thus not limited to pure altruism. The influence of reputations in a gift culture are explored by Eric S. Raymond in [9].

2.2 Submissions, rating, and moderation

The power of the open information pool is also its greatest weakness; the open form makes the information pool vulnerable to errors and abuse. The rating and moderation component ensures that the database is kept clean. Users can submit information into the pool with submission tools. Submissions can be of two types: inserts and modifications. For the storage of modifications within an information pool there are two methods: it is possible to store the original database and the subsequent modifications, or to store only the latest updated version. What method should be used depends on the need for history within the information pool. Different rating and moderation policies can be used for inserts and modifications. For example, an information pool in the form of an encyclopedia for computer science can have the following policy: every Internet user can freely insert new terms and descriptions, and a team of moderators is appointed to modify and delete entries. With such an open policy of an information pool, attacks on integrity and availability cannot be prevented. Write access could be blocked for new users, users only obtain write access after using the information pool for some time. Some boundaries could be set on number of inserts and modifications. When anonymous inserts are not allowed, limits on the number of submissions per user can be set. Another possibility is to limit the number of these transactions per computer by using the IP address. Numerous, slightly different policies can be constructed around an identical central mechanism, several policies can also be mixed to form a new, hybrid policy.

2.2.1 Submitter based

A simple policy is to place the first submitter of the content in control of subsequent modifications. New information may be freely inserted. With every insert the e-mail address is requested. Modifications are allowed but have to be approved by the original submitter, using e-mail. If the original submitter does not respond within a reasonable amount of time, he loses his approval rights. The original submitter is then automatically replaced by the submitter of the modification. This policy is the most simple to use and implement, but it does not offer strong protection against errors.

2.2.2 Reputation based

A more advanced system is build around the reputation of users. Each user is registered with a (nick)name and a number that indicates his reputation. A reputation function calculates a reputation based on several aspects of the user, like the amount of activity of the user, number of retrievals of his submitted information, the number of modifications to his submissions, etc. Based on this reputation all sorts of moderation methods become possible. For example, a high reputation enables people to moderate content submitted by people of significantly lower reputation. This policy requires registration and identification of the user with for example an account and password. This gives a barrier for submission of information, but also more protection. When a new account is requested the password is e-mailed to the user, thus a valid e-mail adress is required. A reputation-based system is more complex, provides a small barrier, and offers stronger protection against errors than the submitter based policy.

2.2.3 Democratic based

With a rating mechanism the majority determines what is considered "correct". Users indicate the quality of every insert and modification. Quality ratings can consist of numbers that indicate the level of accuracy, completeness, or grammatical correctness, it is also possible to rate quality with a "better/worse than" indication. It is very difficult to rate the content of a *web page* because the rating is strongly dependent of the point of view and the purpose of the text, according to [10]. Because a database has more structure and a high level of context, we belief accurate quality rating is possible. When users search through the information pool the ratings can be used to filter out information below the desired quality level. The democratic and reputation based policy can be combined into a system where votes of users are weighted with their reputation, and (dis)agreements influence the reputation of the submitter. A problem with the democratic policy is that users cannot always be objective and independant, for example the majority can determine that false popular belief is true.

2.2.4 Expert based

Experts are (democratically) appointed users that control all the inserts and modifications. Users can freely submit inserts and modification, but the experts determine if they should be added to the open information pool. A variation of this policy is to create a hierarchy of experts. Using the expert policy places a very large responsibility at the experts. Problems may arise if experts do not have sufficient time for moderation, lack of interest for their task, or if their objectivity is questionable. This policy is used within the Open Directory Project [26].

2.3 Databases

A database can contain text, pictures, music, movies, and other data. Examples of possible databases include a history database with people, places and events, encyclopedia on various topics, tutorials on various levels, consumer reviews on products, medical information, product pricing per (e-)store, stock market numbers, TV listings, etc.

Representation of information is a difficult subject. The information stored by the WWW is not structured, hence the context of information is hard to find. Recent WWW developments include adding meta-tags or database extensions that describe the context of WWW pages [6]. When a database is used for storing the information the context is well defined and the content is highly structured [19]. Using a relational database for the storage of web-based information is more powerful than the method of using standard files. The additional power of relational databases is particularly strong when using large pools of structured information.

The demands and properties of a database depend on the class of information stored. The first separation we make is between objective and subjective information. The former is an unbiased truth about subjects, phenomena, people, places, objects, companies, etc. The latter is a person's belief or opinion about some subject, people, etc. with a biased truth. The second separation we make is between deterministic information and non-deterministic information. When there is only one answer and one logical textual representation to a question we call it deterministic information. For example the question "In what year was Napoleon Bonaparte born?" results in a single answer. The question "What is the

life story of Napoleon Bonaparte?”, can be a complete paragraph or book of non-deterministic information.

Objective information is more easy to store and to maintain in the database than subjective information. Moderation is simplified when the content is free of beliefs and opinions. When non-deterministic information is stored in the database, the demands on the rating and moderation tools are higher. Non-deterministic information is very hard to capture and maintain in a value of a field in a database. The reason for this is that modifications on non-deterministic information in text form can be replacements of the whole text, small modifications on several sentences, a new combination of several other suggested modifications, a new structure of the whole text without modification of sentences, etc.

2.4 WWW extraction tools

Open information pools are not replacements of current WWW pages. With WWW extraction tools we can add context and structure to the WWW content and insert it into an open information pool. Extraction tools are very useful to start a new information pool and re-use WWW based content. Several extraction tools exist that can extract the information from a WWW site [15, 18]. Most extractions tools use a *wrapper* for each WWW site. A *wrapper* is a program that extracts information from a specific WWW page and presents this information to the user or inserts it into a database. Because each WWW site is different, wrappers are unique to each WWW site. The extraction tools can generate these wrappers after they are configured for a particular WWW site by the user. This user configuration is time consuming. In [5] software is described that can detect the structure of a web page automatically. This software does not require user configuration, but the software is reported to show “meaningful” results in only 70 % of the cases. Even with these advanced WWW extraction tools, submitters must find WWW sites that contain information missing in the information pools. When automatic WWW extraction tools are used for submission, the submitter information must be present to enable moderation. The extraction tools must therefore be configured with information about the submitter. This, however is not necessary when anonymous submissions are allowed.

2.5 Search tools

Search tools can help the user to search through the information pools. Search tools are more effective than the general Internet search engines because the information is stored in a more structured way, within a known context. In [13] a distinction is made between *discovery queries*, used by WWW search engines such as Yahoo, Altavista, etc. and *retrieval queries* that are queries on a specific collection of WWW pages (intranet) that are well maintained, and have a known structure. WWW search engines often fail to exploit the structure of such WWW pages, because the exact semantics of links remain invisible. Because WWW search engines cannot exploit structure, they prefer indexing “flat” WWW pages.

Discovery queries of Internet search engines ask keywords to make a sorted list of relevant WWW pages. Often these discovery queries result in a large numbers of matching pages. The coverage and recency of Internet search engines (Altavista, Excite, HotBot, Infoseek, Lycos, Northern Light) is extensively analysed in [12]. This study, published in 1998, estimated that the lower bound for the number of WWW pages is 320 million. It is also shown that no single Internet search engine indexes more than one-third of the “indexable WWW”. In contrast, retrieval queries on information pools do not suffer from the effect that information is not “indexable”. A search is broken into two steps. First, the appropriate information pool is located. Second, this single database is searched to answer the query.

2.6 Distribution tools

For performance and reliability the information pools can be replicated. The openness of the information pools also dictates that there must be no single person or organisation in control. Information must be freely shared among all interested users on the Internet. Users must be able to download a whole database and access it off-line. Updates must be exchanged between different sites to keep them synchronised. Updates must not be controlled by a single site in a scalable distributed system. There are several update mechanisms. (1) Updates can be distributed in the form of a chain where each location forward the received updates and his own updates to the next location. (2) A hierarchical system

can be used distributes the updates from a group of top level systems. (3) The use of a news group to broadcast the updates. A news group with formatted e-mails is a particular useful method of distributing updates because it builds on a world-wide broadcast mechanism. The reliability of a system using news groups is higher than the alternatives, yet more bandwidth and other resources are used.

A nice property of our information pools is that they are self-describing. Having a description of the databases off all information pools is very useful. A special database called *meta* is used to describe the various databases, the policy they use, the tables and fields therein, the distribution method, and the locations where they are stored. The open meta database has a special moderation method. The inserts can only be made from the site which hosts the open database. This can be simply checked using the IP number. The meta database is useful for search tools because it contains the context of all information.

3 Legal issues

Copyright laws on the Internet have always been under pressure because of the ease of duplication. Redistribution and publication of copyrighted material is in most cases not allowed. Copyrighted material *cannot* be entered into an open information pool without permission of the copyright holder. For a good introduction about copyright laws and databases, see [2, 14]. It is impossible to claim the copyright of facts, ideas, and public information. The non-trivial question is, what material is copyrighted, and what is an unprotected fact, idea, or public information.

The best legal term describing our information pool concept is the term "automated database". The copyright law of the United States defines an automated database as: "*a body of facts, data, or other information assembled into an organised format suitable for use in a computer and comprising one or more files*" [11]. The copyright laws of the US and many other countries only protect "original work". The definition of an original work was traditionally coupled to some form of *creativity* within this work. However, it is very hard to show the creativity for the content of an automated database. To extend the copyright protection to

factual automated database the "sweat of the brow" doctrine was introduced. But this doctrine was recently abandoned in a US supreme court ruling of *Rural telephone Service Co., Inc v. Feist Publications, Inc.* 663 F. Supp. 214 218 (1987). Open information pools of facts and public information can therefore not be copyrighted because there is no original work, no creativity, and the "sweat of the brow" doctrine was abandoned. The implications for the open information pool concept are that making a copy of an existing database on the WWW with facts, ideas, and public information is allowed, but copyrighted information remains off-limits.

4 Implementation

In this section we first discuss measurements we conducted on rating and moderation tools. Second, we discuss our own prototype that is the first step towards a full implementation of our open information pool concept.

4.1 Rating and moderation measurements

Rating and moderation tools have been discussed extensively in the literature, see for a good example [10], and for an overview [4]. However, we were unable to find any publication on the performance of such tools with both *real* content and a large group of *real* users. To investigate the dynamics of rating and moderation tools, and to support the validity of our open information pool concept we conducted measurements on a web site with operational rating and moderation tools. We choose the Slashdot.org web site because it has already a lot of the elements we use in our open information pool concept. Slashdot.org shows important news items on various topics in computer science and gives readers the ability to freely attach *news comments* with remarks, enhancements, and insights to a news item. Inserted news comments can be viewed by all readers and are attached to the original news item. All news items and comments are stored in a large relational database. Slashdot.org uses a hybrid rating and moderation system that is based on the reputation and democratic mechanisms discussed in Section 2.2. However, Slashdot.org is *not* generic, it is fully dedicated to the "news forum" purpose. The

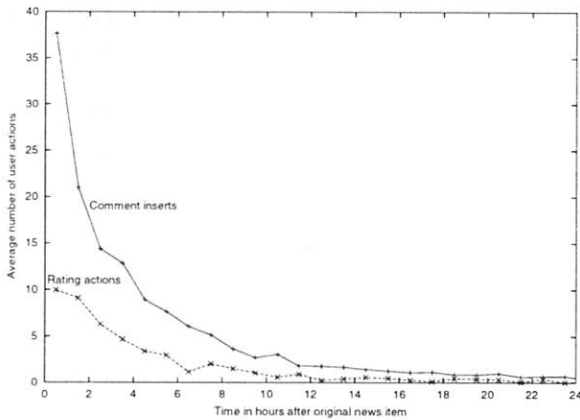


Figure 2: Slashdot.org user actions.

database routines, rating system, and moderation system cannot be re-used for other classes of information besides news items. Another important limitation is that news comments cannot be modified once inserted. The lack of modification options is not important for news comments, but does prevent news comments from being corrected for spelling errors. A generic implementation of an open information pool must allow several operations on the database content to update or improve it. When the content is text, the operations are for example correcting spelling errors, joining several paragraphs together, making modifications on several sentences, etc.

Within Slashdot.org every comment to a news item is rated with a *score* from -1 (inferior) to 5 (insightful) by the readers. The initial score of a news comment is 1 by default, 2 for users with a high reputation, and 0 for anonymous news comments. A percentage of the (randomly-selected) readers with sufficiently high reputation are allowed to influence the scores. When users are asked to influence the score they can add or subtract a single point to the score of a news comment. The reputation of a user is calculated based on the score of his recently posted news comments. Readers can set the level of moderation by filtering news comments below a score threshold. If the lowest threshold of -1 is selected, all news comments are visible.

In Figure 2 two user actions are shown: the number of new added news comments and the number of rating actions. The news comments are the average number of inserted comments per hour in the discussion of a single news item. The time is set relative to the publication of the news item. Each

Comment score	Percentage
inferiour, -1	8.1 %
0	31.0 %
1	40.3 %
2	14.5 %
3	3.8 %
4	1.2 %
insightfull, 5	1.1 %

Table 1: Slashdot.org comment score distribution.

rating action is the addition or subtraction of a point from the news comment score. The results are calculated from an analysis of 30 news items which received more than 4,250 comments and were subject to 1,400 rating actions. The average number of inserted comments per news item is about 142 (4250/30). This figure shows that the activity of the users is fairly high and fast, on average half of the comments is inserted within three hours of the news item's release. Within the first hour of the news item release almost 38 news comments are inserted. After this initial attention the user activity drops fast and after 12 hours less than two comments are inserted per hour. The rating system does not show this sharp decrease. On average almost 10 rating actions are performed within the first hour on the 38 inserted news comments. After 10 hours, less than one rating action is carried out per hour.

The resulting distribution of the news comments scores is shown in Table 1. If a reader selects the threshold of 3 for reading, 93.9 % of the news comments are not shown. Unfortunately it is very difficult to determine objectively if the remaining high ranking comments are really of high quality. It is the personal opinion of the author that the score of a news comment gives a good indication of the quality of the comment. Deviations between the score and the quality are not frequent and are seldom more than a single point. Overall the rating and moderation system works effectively. For example, news comments containing strong language with no insight, no intelligent remarks, and no enhancement are quickly rated -1, (inferior).

The required time for a news comment to obtain the final score is shown in Figure 3, for clarity only the four major start and end score combinations are shown. Each change in the score is a result of a user rating action. Time is taken relative to the insertion of the news comment. The number of measured news comments (n) with this start and end score

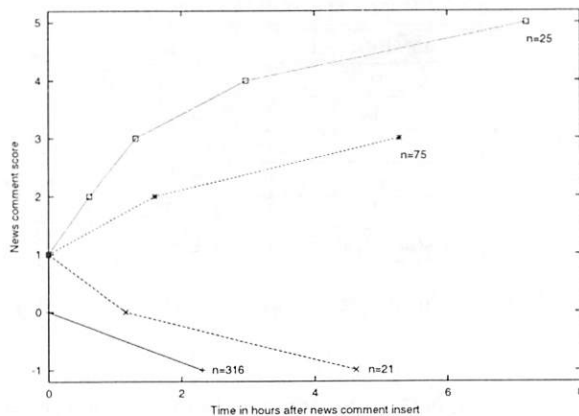


Figure 3: Speed of the rating system.

combination is shown at the endpoint of the lines. It is important to know that the initial score is changed for only 21.8 % of the inserted news comments. On average it takes 37 minutes for news comments with an initial score of 1 and final score of 5 to receive the first addition to the score. For news comments with an final score of 3 the first addition comes after 1 hour and 36 minutes. Receiving the last point towards the final score takes significantly longer. For the following [initial, final] score combinations [1,5], [1,3], [1,-1], [0,-1] the last rating point is received after 253, 220, 207, and 138 minutes respectively. This indicates that the convergence towards the final score starts fast and then slows down.

To summarise we note that the rating of the news comments by the users works effectively. The users are very active with both inserting comments and rating comments. The rating system of Slashdot.org works on a surprisingly small time scale; insightful news comments in the database get their first reward after only 37 minutes. With the measurements on Slashdot.org we see that subjective, non-deterministic information (see Section 2.3) can be rated fast and accurately. Another important characteristic is that users are free to determine the level of moderation by setting a filtering threshold.

4.2 Prototype

We are currently building a prototype of a system for a single open information pool, with all the elements of the overall structure outlined in Figure 1. Our first goal is to develop new rating and moderation tools from studies with actual users and real content that work effectively for various classes

of information and allow all sorts of modifications. The next step is to build a generic implementation that enables a user to create a new open information pool using only a WWW interface. With this WWW interface the user selects a rating method, moderation method, tables, fields, and other options. The system would automatically create the information pool and insert this new pool into the meta-database (see Section 2.6) and allow all Internet users to browse and submit information.

For the database content of the prototype we have chosen music information. The database consists of music artists/band biographies, produced audio CDs, audio CD cover pictures, songs on every audio CD, lyrics of the songs with timing information (\approx karaoke), and MIDI files. There are four reasons for this decision. First, this content has a high degree of context, requires frequent updates, and combines various information classes (see Section 2.3). This is important because we want to study rating and moderation systems for all information classes in action. Second, the content of this database is attractive for a very large and active user group. Third, because the existing database with some of this information called "Audio CD database" [21] is taken from the public domain, see Section 1, we want to return this database to the public domain. Fourth, we can re-use and extend the source code of the CDIndex project [20]. The open source CDIndex project is developing software for an *open* audio CD database as an alternative for the *closed* audio CD database. The aim of CDIndex is the same as the original audio CD database, focused to storing audio CD "table of contents". The table of content of an audio CD consists of the names of the songs and artists, this information is not present on the CD itself. CDIndex has an operational database and produced working code to anonymously insert information. Rating, moderation, and distribution tools are not yet developed and the database is virtually empty.

Our prototype is implemented in Perl on a Linux system using the MySQL database and Apache WWW server. The music information database of the prototype is finished and can be viewed, searched or filled from an ordinary browser. Submission, voting, and moderation tools are still under development. We implemented a generic search tool that can be used for keyword searches and to browse through a database. This generic search tool queries the relational database for tables and fields that can be searched. This information is not present

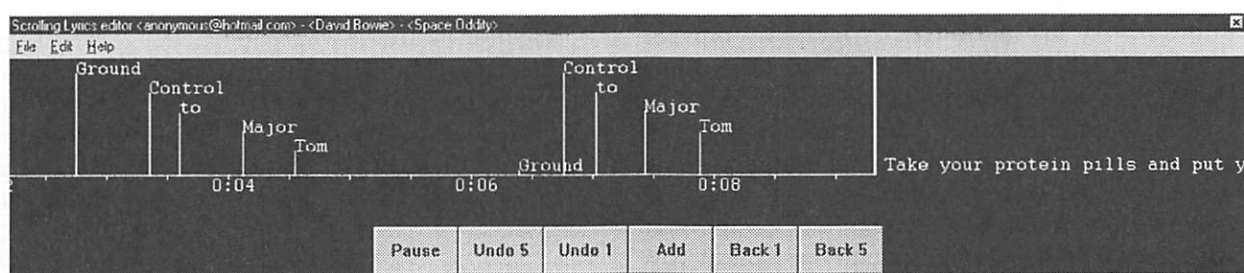


Figure 4: The Scrolling Lyrics creation and submission tool.

in the generic search tool. The search tool uses the database structure information to show a HTML search form. A basic WWW extraction tool has been made. This WWW extraction tool was configured to extract audio CD cover pictures, MIDI files and lyrics from several WWW sites. The open information pool now contains more than 10,000 audio CD covers, numerous MIDI files, and an extensive amount of lyrics.

A specific client program called "Scrolling Lyrics" is developed for the display, creation, and submission of the lyrics with timing information. Scrolling Lyrics is an Open Source plug-in for the popular Windows95 music player called "WinAmp". A screen shot of Scrolling Lyrics program in edit mode is shown in Figure 4. A user can enter the lyrics, add a timestamp for every word in the song, and submit this information to a server of their choosing. In the screen shot, text with time stamps is shown as well as plain text. When the music is playing users can click the "add" button to add a timestamp to a word in the song. The text with timestamps has a staircase effect for readability. When the scrolling lyrics program is in play mode, the lyrics of the song scroll in synchronisation with the music. Options are included to save this scrolling lyrics information inside a compressed MPEG audio file (mp3) and to submit it for entry in the music database. Submissions are formatted in XML, a more general form of HTML. The submissions are delivered to a server with the HTTP Post method. The processing is done with perl CGI scripts, they insert the submission in the music database. The distribution tools can optionally post every submission that arrives at the server directly in a news group.

There are several unsolved legal issues tied to this music database. The lyrics of almost all popular songs are copyrighted and cannot be entered into the music database without permission. If the scrolling lyrics information is a "new and original work" it

is not copyrighted and can be entered freely. The question of copyright on scrolling lyrics information is unresolved. Under the local Dutch copyright legislation it is permitted to give people a personal copy of lyrics without violating the copyright. Using scanned images of audio CD covers is also permitted, provided that they are used for on-line sales catalogs, for example an electronic audio CD shop with a preview image of the audio CDs on sale.

The source code of our prototype and the scrolling lyrics program is freely available from [22] under the Open Source license. The measurement software for the Slashdot.org rating and moderation tools is available on request. This measurement software can interfere with the proper operation of the Slashdot.org server.

5 Related work

We have not been able to find any earlier work raising our basic idea - a world-wide collection of open, public-writable databases with moderation and world-wide distribution. However our idea touches on a number of other publications in different parts of computer science, including hypertext, databases, knowledge bases, etc.

Various researchers have investigated the combination of relational databases and the WWW. In [6] the idea of including the database tables, attributes, and relations inside HTML was presented. The complete separation of content and appearance is argued in [19], where the content is organised as a database. The developers of RCS and CVS [8] have created tools to update, synchronise, and distribute files. These tools provide some basic support for public write access and moderation of submissions coordinated by a central server. CVS op-

erates on the file and directory level and provides an abstraction level that is very basic, for example database records are outside the scope of CVS. Our concept is different from CVS because we use a distributed structure, operate on the database level, and use generic rating and moderation tools. As early as 1990 the first prototype browser appeared with both read and write capability, yet the WWW has primarily been used as a read-only tool. The recent WebDAV standard [16] transforms the read-only WWW into a writable, collaborative medium. Access is restricted and cryptographic authentication schemes are used to enforce this. WebDAV defines mechanisms for file locking, version management, hierarchical organisation, and access control. The standard does not specify distribution, rating, and moderation tools. It differs from our approach because it is designed only for use by a closed group of users and cannot be applied on a global Internet scale. Annotation and rating of WWW pages has been explored in [4, 10]. The annotation tools proposed in these papers use separate annotation servers and are limited to adding comment to existing WWW pages. This differs from our approach, we define integrated rating and moderation tools for structured information. The controversial US Communications Decency Act [7] increased the interest in such content rating and moderation systems. The Usenet system has some similarities to our concept. Usenet is also fully distributed and has public write access, but it does not have standard rating tools and enforces no rigid structure. The hierarchy of news groups with different subject areas is very loose when compared to the rigid structure of a relational database.

6 Conclusions and future work

We have introduced the idea of structured information pools that are open to the public. By giving public write access, information pools grow with each submission. We are working on a generic open information pool implementation based on a web server and relational database system. With this implementation a number of WWW drawbacks would be removed. Generic rating, moderation, distribution, and WWW extraction tools are needed to fill these open information pools and to keep them clean, accessible, and reliable. With our system we want to counter the trend that information is taken from the public domain.

In the near future we will conduct a long term study to determine the dynamics of a large scale and intensively used open information pool to refine our implementation. We hope that our next implementation will be another step in the realisation of the dream that all knowledge of humanity will be unlocked.

Acknowledgements

The author wishes to thank Clem Cole and Koen Langendoen for their valuable advice, and the anonymous reviewers for their good suggestions.

References

- [1] Bush V., "As We May Think", Atlantic Monthly, July, 1945.
- [2] Barlow J.P., "The Economy of Ideas: A framework for rethinking patents and copyrights in the Digital Age", wired, March 1994.
- [3] Barrie J., Presti D., "World Wide Web as an instructional tool", Science, 274, 371-372, October 18, 1996.
- [4] Bouvin N.O., "Unifying strategies for web augmentation", ACM Hypertext '99 conference Doctoral Consortium, February 1999.
- [5] Cohen W., "Recognizing structure in web pages using similarity queries", Proceedings of the Sixteenth National Conference on Artificial Intelligence, 1999.
- [6] Dobson S.A., "Lightweight databases", 1st Workshop on Logic Programming Tools for INTERNET, September 1996.
- [7] Exon J., "Communication Decency Amendment", US Senate, July 1995.
- [8] Grune D., "Concurrent Versions System, a method for independent cooperation", technical report 113, Vrije Universiteit, Amsterdam, 1986.
- [9] Raymond E.S., "Cathedral & the Bazaar", Sebastopol California, O'Reilly & Associates Inc, ISBN 1-56592-724-9, www.tuxedo.org/~esr/writings.

- [10] Roscheisen M., Winograd T., Paepcke A., "Content rating and other third-party value-added information defining and enabling platform", CNRI Journal D-Lib, August 1995.
- [11] United States Copyright Office, "Copyright Registration for Automated Databases", publication Circular 65.
- [12] Lawrence S., Giles L., "Searching the World Wide Web", Science, Vol. 280, Num. 5360, page 98, 1998.
- [13] Lim E.-P., et.al., "Querying structured web resources", Proceedings of the 3rd ACM International Conference on Digital Libraries, Pittsburgh, Pennsylvania, June 1998.
- [14] Losey R.C., "Practical and legal protection of computer databases", Orlando, Florida, 1995, FloridaLawFirm.com/article.html.
- [15] Mecca G., et.al., "The Araneus web-based management system", in exhibits program of ACM SIGMOD, 1998.
- [16] Whitehead E.J., Wiggins M., "WebDAV: IETF standard for collaborative authoring on the web", IEEE Internet Computing, page 34 - 40, September - October 1998.
- [17] Nelson T., "Xanalogical Media: needed now more than ever", *to appear in* ACM computing surveys, hypertext issue.
- [18] Sahuguet A., Azavant F., "W4F: a WysiWyg Web Wrapper Factory", Technical Report, University of Pennsylvania, 1998.
- [19] Sandewall E., "Towards a world-wide database", 5th International WWW conference, May 1996.
- [20] The open CD database, www.CDIndex.org
- [21] The audio CD database, www.cddb.com
- [22] The ultimate music database, www.mp3.nl
- [23] Common vulnerabilities and exposures list, cve.mitre.org
- [24] Archarology index, www.openarchaeology.org
- [25] Open content initiative, www.OpenContent.org
- [26] Open directory project, www.dmoz.org
- [27] Link everything on-line, www.leo.org
- [28] Internet movie database, www.imdb.com
- [29] Open hardware design, www.lart.tudelft.nl

The GNOME Canvas: a Generic Engine for Structured Graphics

Federico Mena-Quintero

Helix Code, Inc.

federico@helixcode.com

Raph Levien

Code Art Studio

raph@gimp.org

Abstract

The GNOME canvas is a generic engine for structured graphics that offers a rich imaging model, high performance rendering, and a powerful high-level API. Application programmers can use the canvas to create interactive graphics displays easily. Many GNOME applications use the canvas as their main display engine, some of them using the basic functionality provided by the canvas, and others by extending it for their particular needs. This paper describes the architecture of the canvas in detail and examines the way it is used in several GNOME applications.

1 Introduction

The GNOME canvas is a generic, high-level engine for creating structured graphics. A canvas is a window that contains a collection of graphical *items*, including lines, polygons, rectangles, ellipses, and text. The term *structured graphics* means that you can place these graphical items in the canvas and refer to them later to change their attributes. For example, a program could place a white rectangle at some specific position, and later in its execution it could change the color, position, or any other attribute of the rectangle. The canvas would then take care of all redrawing operations.

Items inside a canvas can be organized in a tree of nested *groups* which are nodes in the tree, and terminal items which are leaves in the tree. The canvas allows arbitrary affine transformations like rotation, scaling, and translation to be applied to items and groups; if a transformation is applied to a canvas

group, then all of its children will be transformed accordingly. This tree organization makes it easy to create hierarchical drawings.

The GNOME canvas has an open interface that allows applications to create their own custom canvas item types. This means that the canvas can work as a generic display engine for applications. One of the following sections in this paper describes case-by-case examples of the use of the canvas in different GNOME applications.

The canvas has multiple rendering backends, one for rendering using GDK to plain X drawables[4], one for rendering using high-quality antialiasing and transparency, and one for sending the contents of the canvas to a printer.

This paper describes the architecture of the canvas and its high-quality imaging model, and presents some examples of the use of the canvas in different GNOME applications. It also describes some of the future directions for development of the canvas display engine.

2 Architecture of the Canvas

The GNOME canvas was originally based on the canvas widget in the Tk toolkit[9], which is in turn based on Joel Bartlett's *ezd* program, which provides structured graphics in a Scheme environment. The main enhancements that the GNOME canvas provides to the original Tk design are integration with the GTK+ object and signal/slot system[5], nested groups of items, generalized affine transformations, and a high-quality antialiased rendering mode. From the standpoint of the user, the can-

vas presents the following characteristics:

- The user is able to create graphical items like lines, boxes, ellipses, and text, place them on the canvas, and refer to them later for manipulation. The attributes of an object can be changed at any time; these include the color of the item, its line style, and its position.
- Canvas items receive events just as if they were normal X windows or other GTK+ widgets. Applications receive these events with the normal signal/slot system in GTK+. An application can then connect to the event signals of its canvas items and define the particular behavior it requires. Common actions include moving an item when the user clicks and drags it with the mouse, or highlighting an item by changing its color when the mouse pointer enters the area occupied by the item.
- Items can be grouped together in canvas item groups, and these groups can be nested within each other to form a tree structure. A canvas has a single root group which does not have a parent. Operations such as deleting or moving a group apply to all its items, thus making it easy to create hierarchical drawings. It also has performance advantages since the canvas can use recursive bounding boxes to cull out items for drawing and hit testing operations.
- Items support arbitrary affine transformations, so they can be translated, scaled, rotated, and sheared in any way. Affine transformations apply to item groups as a whole, so the children of a group will obtain the same base transformation as its parent.
- The zooming factor of the canvas can be changed at any time, and the canvas will handle all scrolling issues by itself.
- The canvas takes care of all drawing operations so that it never flickers, and so that the user does not have to worry about repainting the items he wants to display.

2.1 Canvas Items and the GTK+ Object System

Canvas items are GTK+ objects derived from an abstract `GnomeCanvasItem` class, which defines several methods that all items must implement. These

methods are used to perform drawing, hit testing, and updating of items when their attributes change.

Using the GTK+ object system for canvas items provides several advantages:

- No extra work is involved in wrapping them for different language bindings, since GTK+ objects and their attributes are wrapped automatically by most bindings.
- Canvas items use the usual signal/slot mechanism to emit events, making it easy for the programmer to define behavior for the items.
- One can associate arbitrary data items to canvas items, via the GTK+ dataset mechanism.

All the attributes of canvas items, like color, position, and line style, are configured and queried using the GTK+ object argument system. Canvas items may have many configurable attributes, so using the argument system allows us to minimize the number of API entry points, and also makes it easy to write language bindings for the canvas and its items — all canvas item attributes can be configured using a single function call.

2.2 Grouping of Items

Items in the canvas are organized in a tree hierarchy. Items can be groups, which are nodes in the tree, or terminal items, which are leaves in the tree. Groups can contain any number of children, which can in turn be terminal items or other groups. Items can thus be nested to an arbitrary depth inside the canvas, making it easy to create hierarchical drawings.

A canvas has a single root group. For very simple drawings or diagrams, the programmer may want to put all items directly under the root group. For more complicated, structured drawings, it will be convenient to create a hierarchical organization — a circuit editor may want to represent an adder as a group of basic logic gates, which in can in turn be groups of primitive canvas items like lines and rectangles.

The bounding box of a canvas group surrounds all of its children, so drawing and hit testing operations

can be made more efficient by recursive culling of items.

Items inside a group are stacked on top of each other, and items that are higher up in the stack obscure the items below them. The canvas provides functions to change the stacking order of items by raising or lowering them within their parent group's stack.

2.3 Behavior of Items

The canvas does not have any predefined behavior for items. Instead, the programmer will connect to the event signals of the different canvas items, capture events from the user, and define whatever behavior is appropriate to the application.

When an event signal is emitted for an item, it is propagated up the item hierarchy and re-emitted for its parent groups until one event handler marks the event as 'handled'. This allows the user to treat a group of items as a single meta-item; only a single signal connection is required to receive events from any of the items in a group.

Items can receive the following events: button presses and releases, pointer motion events, key presses and releases, focus in/out events, and mouse enter/leave notifications. Thus, items are very similar to normal GTK+ widgets or X windows from the programmer's point of view.

2.4 Delayed Update/Redraw Model

One of the goals of the canvas is to eliminate flicker when drawing items. Flicker is caused when an area is repainted multiple times with different colors; for example, a stack of colored rectangles would flicker if it were painted directly to the screen, one rectangle after another.

The canvas solves this problem by using a special form of double-buffering. When an area of the canvas needs to be repainted, the following actions take place:

1. The canvas creates a temporary, offscreen pixmap with the same size as the area that needs to be repainted.

2. If an item's bounding box intersects the area that needs to be repainted, the canvas asks the item to paint itself to the pixmap created in (1). The canvas thus walks the tree of items in the normal Z-order.
3. The canvas does a bitblt of the pixmap to the screen and destroys the pixmap.

The visual effect is that the whole area is painted simultaneously, eliminating all flicker.

This process actually takes place during the idle loop of the application. Repainting the canvas in the idle loop means that at that point all interaction between the application and canvas items is finished, i.e. the application literally has nothing else to do, so it is appropriate to flush all pending redraws.

2.4.1 Delayed Updates

We have not explained how the canvas actually figures out the area that needs to be repainted. Let us consider a few simple cases:

1. A solid-colored rectangle with dimensions (w, h) is translated with offsets (dx, dy) such that $|dx| < w$ and $|dy| < h$. In this case, the minimal redraw area consists of two L-shaped regions that are the symmetric difference between the old and the new rectangles (see Figure 1).

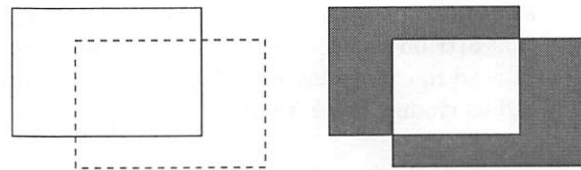


Figure 1: When a solid-colored rectangle moves, the redraw area consists of two L-shaped regions.

2. A solid-colored rectangle changes color. In this case, the whole area occupied by the rectangle needs to be redrawn.
3. A solid-colored circle changes radius. The minimal redraw area thus consists of the donut-shaped region which is, again, the symmetric difference between the old and new circles.

4. Some characters in a string of text are changed, for example, when the user is editing a label. If the string is drawn using a monospaced font, the minimal redraw area consists of the area occupied by the characters that changed. If the string is drawn using a proportional font, the redraw area will be more complicated.

One of the goals of the canvas is to compute the minimal redraw area for each operation. This is important because we wish to make the final bitblt operations as small as possible; experiments have shown that memory bandwidth tends to be a bigger problem than CPU speed.

Canvas items have two important methods, `::update()` and `::draw()`¹. The former is responsible for calculating the area that needs to be redrawn when the item's attributes change, and the latter gets called when the final redraw area has been computed by the canvas and an item needs to paint itself.

The full sequence of operations that starts when an item's attributes are changed and ends when the canvas paints to the screen is as follows:

1. A state change happens in a canvas item, usually from direct manipulation through the user interface. An item may thus change attributes like color or position.
2. The canvas item stores an internal flag saying that it needs to change one of its attributes, and also stores the necessary information to change that attribute. For example, an item may store "I need to change my fill color to blue", or "my radius changed to 7.5 units".
3. The canvas item then queues an update from the canvas using the `gnome_canvas_item_request_update()` function. The canvas installs an idle handler on the GTK+ main loop.
4. The application keeps running, possibly requesting attribute changes from other items, until it finishes its work and all its interaction-related tasks, and gets back to the idle loop.

¹Items actually have `::draw()` for canvases in GDK mode, and `::render()` for canvases in antialiased mode. However, both methods are responsible for drawing the item, so we will refer to them generically as "`::draw()`".

5. The idle handler for the canvas is run. The canvas calls the `::update()` method of each item that requested an update.
6. The `::update()` method for an item flags the item as no longer needing an update. It should recalculate the item's internal state based on the flags set in step 2, for example, by changing colors or line styles in an X graphics context (GC). This method is also responsible for recomputing the item's bounding box if the item's bounds changed. Then, it should queue a redraw from the canvas based on its new state. We will later describe how this area is represented.
7. After all items that need it have been updated, and as such they have recomputed their bounding boxes and queued the appropriate redraws, the canvas calls the `::draw()` or `::render()` method of items that need it. This method is passed a temporary pixmap in the case of a canvas in GDK mode, or an RGB pixel buffer in the case of an antialiased canvas.
8. The canvas is now fully updated and redrawn, and the application continues to run.

As we have seen, the `::update()` method is responsible for doing housekeeping work like changing GC colors and line styles, recomputing an item's bounding box, and queueing the proper redraws. This method is called from the canvas' idle handler. The reason for delaying GC updates and the like to the idle loop is that an application may change many attributes of a canvas item before getting back to the idle loop; if the item changed GCs or recomputed its bounding box for every time an attribute was changed, this could turn into a performance problem, since many such operations are expensive. Delaying all the updating work until the idle loop means that the application's interaction with items has finished, so the items know their final state at that point and can compute the most efficient way to do their respective updates.

3 The Libart Imaging Model

Up to now we have discussed the way the GNOME canvas operates internally. In this section we will describe the imaging model the canvas supports.

Libart is a library that provides a superset of the PostScript imaging model[2], and it extends it with support for antialiasing and alpha transparency. This means that the edges of graphics primitives such as Bézier paths are smoothed out to eliminate jaggies. Also, such primitives and images can be rendered and composited together using transparency information.

Libart is quite similar in design and scope to such “next-generation” imaging models as Adobe’s Bravo[1], the Java 2D API[10], Adobe’s Precision Graphics Markup Language (PGML)[11], and the W3C’s Scalable Vector Graphics Specification (SVG)[12].

The GNOME canvas uses Libart to render its primitives when it is in antialiased mode. Also, it uses Libart’s microtile arrays, described below, to represent the areas that need redrawing.

The following sections describe the main features of libart.

3.1 Vector Paths

Libart’s vector paths are built from the familiar PostScript opcodes such as `moveto`, `lineto`, and `curveto`. Paths can be composed of multiple closed sets of segments and thus have holes in them. Paths can also cross themselves any number of times.

3.2 Sorted Vector Paths

A sorted vector path, or SVP, is a processed version of a normal vector path such that it satisfies the ‘nocross’ property, that is, it does not have crossing segments and retains the same winding number as the original vector path. Also, its segments are sorted so that they have monotonically-increasing *y* coordinates. This allows for very efficient rendering, since the segments can be traversed in scanline order.

3.3 Antialiased Rendering

Sorted vector paths can be rendered using “perfect resolution”, as opposed to the common technique

of rendering a higher resolution bitmap and averaging down. The SVP precomputation step is important because it allows a vector path to be rendered multiple times very quickly; canvas items compute their vector paths in the `::update()` method, convert them to SVPs, and store these so that they can re-render the SVPs quickly if needed.

3.4 Outline Stroking

This is the computation of stroke outlines for vector paths. The standard PostScript technique is to render each segment of the stroke separately, using small “miter joints” added at each corner. However, this technique is not ideal for antialiased rendering because the resulting number of polygons is large, and the adjoining polygons can produce seams and other artifacts.

Libart creates stroke outlines by computing inner and outer contours around the original vector path, and then performing a boolean union on them. This union operation cleans up any intersections or overlaps of the stroke. The result is a pure vector path that can be efficiently rendered with the usual algorithm. This is faster and visually more precise than rendering each segment of the stroke separately.

The stroking algorithm supports the same line join and cap options as PostScript.

3.5 Vector Path Operations

Libart can perform intersection (clipping), union, difference, and symmetric difference of vector paths. The latter is especially important for exact computation of redraw areas in the canvas.

3.6 Raster Images

Libart supports affine transformations for raster images, so they can be scaled, rotated, and sheared in any way. It also supports full alpha transparency for images and special gamma correction for opacity.

3.7 Microtile Arrays

An important requirement for the canvas is to have an efficient representation of the area that needs to be redrawn. This area can be disjoint and potentially very complex, since items can be scattered across the canvas area and one would wish to avoid painting a single large bounding box for all of them.

Microtile arrays are a simple data structure for representing 2D regions, suited to representing redraw areas.

The array divides the area into a grid aligned on 32-pixel boundaries. Within each grid square is a bounding rectangle, the microtile. Since all coordinates in the microtile are in the range $[0, 32]$, 8 bits are more than sufficient for each coordinate. Since each microtile requires four coordinates to represent its bounding rectangle, each microtile can be conveniently represented with a 32-bit value.

Figure 2 presents the microtile representation of a complex area. The polygon defines the covered area. The shaded region represents the individual microtiles; each microtile is the bounding box within a grid square that surrounds the covered area. For the final redrawing operation, adjacent microtiles are combined together into bigger rectangles, shown as thicker outlines in the diagram. Each of these rectangles is then calculated and drawn to the screen.

Microtile arrays have many advantages which make them ideal for the canvas widget. First, the data structure is compact; a microtile array for a 640×480 window requires only 1200 bytes. Second, it can be manipulated very quickly; the sample polygon in Figure 2 is calculated in about 1.2 ms on a 233 MHz PII. Third, the resulting microtile array is easily decomposed into rectangles.

Rectangle decomposition has several desirable properties, including a bounded number of rectangles — no more than 300 for that 640×480 image, no matter the complexity of the area. Also, rectangles tend to align on 32-pixel boundaries, which can speed things down the rendering pipeline.

In the context of the GNOME canvas, items can queue their redraw areas in any of three ways: they can specify an explicit microtile array, in which case it is added to the canvas' current redraw area; they can request that a rectangular area be redrawn, and

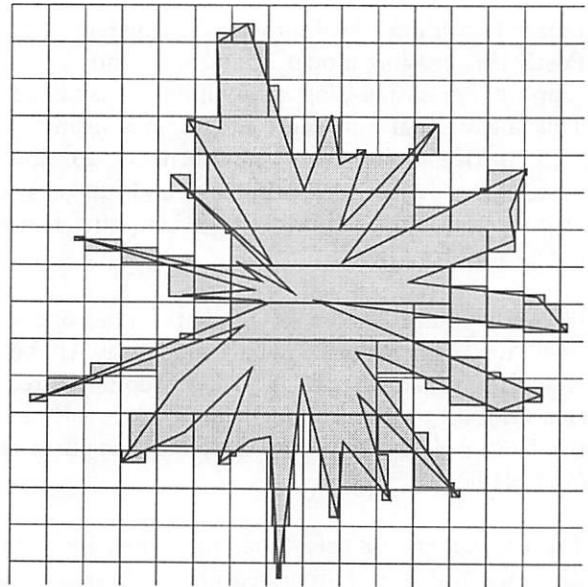


Figure 2: Microtile array that represents the area occupied by a complex polygon. Each little bounding box, or microtile, fits within a grid square. Microtiles are later coalesced into bigger rectangles suitable for redrawing.

so the canvas converts that rectangle to a microtile array; or they can specify a sorted vector path as the redraw area, which is again converted to a microtile array.

3.8 Miscellaneous Utilities

Libart provides miscellaneous functions to handle affine transformations, points, and rectangles. Affine transformations can be applied to vector paths or raster images. Rectangles are used to represent bounding boxes and other things that are useful to the canvas and applications in general.

3.9 The Updating and Rendering Pipeline

Figure 3 presents an illustration of the updating and rendering pipeline of the canvas. The steps are as follows:

1. The first step is to cause a state change in a canvas item, usually from direct manipulation through the user interface.

2. Identify the deltas, or the parts of the display that have actually changed. This is done in the `::update()` method of canvas items.
3. Represent the deltas as a microtile array. Each microtile is a small rectangle that needs updating.
4. The microtile array is decomposed into bigger rectangles for more efficient redrawing.
5. Fifth and sixth, repeated for each rectangle from (4), canvas items are rendered in their normal stacking order, culling them against the bounding boxes defined in (4), and are displayed in the canvas window.

4 Applications that Use the GNOME Canvas

This section describes how different GNOME applications use the canvas as their display engine.

4.1 Gnumeric

Gnumeric is the GNOME spreadsheet program[7]. It uses the canvas as its main display engine to allow for easy event handling, extensibility through components, and flicker-free display.

Gnumeric defines several custom canvas item types:

- An `ItemGrid` item which takes care of displaying all the cells in the spreadsheet. This draws the actual grid and the cell contents, with support for different fonts and colors.
- An `ItemCursor` item that takes care of displaying the specialized selection and active cell combo ‘cursor’, as well as its decorations. The selection has a thick outline which the user can drag to move cells around, and it also has a little rectangle that can be used for the auto-fill function.
- An `ItemBar` item that displays row and column headings for the spreadsheet. The user can drag the edges of the “buttons” that represent rows and columns to resize these. The user

can also click and drag on the buttons themselves to select whole rows or columns in the spreadsheet.

In addition, Gnumeric uses some of the primitive canvas item types such as rectangles, ellipses, and lines to display miscellaneous elements of the user interface.

4.2 Gnome-PIM

Gnome-PIM is the GNOME personal information manager, which consists of a calendar and a contact manager or addressbook.

The calendar program needs to present many interactive graphics displays, such as monthly calendars with captions for appointments, yearly calendars with marked busy days, and other views for weeks and days.

The contact manager program needs to display a familiar representation for “business cards”, or the data that describes a personal contact.

Instead of drawing all of these displays by hand, Gnome-PIM uses the canvas to create these displays out of primitive canvas items such as lines and rectangles. This allows the application to invest more effort in event handling to give the best possible experience to the user, while leaving all the tricky display issues to the canvas.

4.3 Evolution

Evolution is the next-generation mail and groupware program for GNOME. The mailer requires many complex displays such as a hierarchical view of mail folders with previews of the first few lines of mail messages; the rest of the PIM-related modules require calendar and business card displays.

Evolution defines an `ETable` canvas item that implements a model/view/controller abstraction for the display of tabular data. Custom cell renderers can be plugged into this item, turning it into a general-purpose grid display. The canvas allows this complex item to do flicker-free display easily.

Some of the information displays used in Evolution

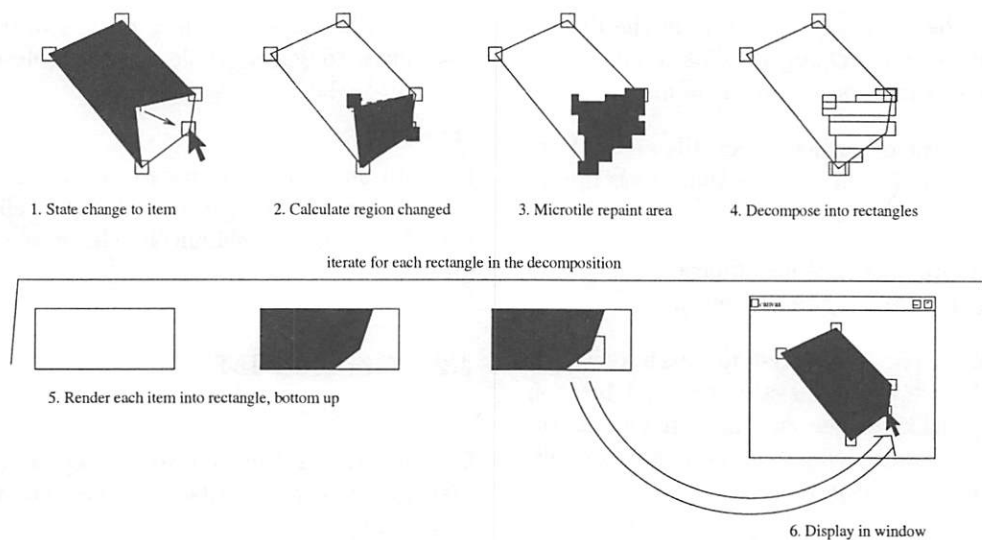


Figure 3: The updating and rendering process

are very complex, as they need to present the user's personal information in a convenient way. The canvas allows Evolution to concentrate on presentation and user interface issues rather than mundane tasks such as redrawing and event handling.

4.4 Eye of Gnome

The Eye of Gnome program is the GNOME image viewing and cataloging program. It uses the canvas for its main image display, and it defines a custom canvas item that can do extremely fast scaling of images suitable for an image viewer.

EOG also defines a model/view/controller abstraction for "wrapped lists", and in turn implements an icon or thumbnail view for large sets of images. It uses special techniques so that only the icons and captions that fit in the canvas window actually exist as canvas items; these are created and destroyed on the fly as the icon list is scrolled and updated.

The canvas allows for easy event handling, and with the delayed update model, also allows for on-the-fly loading and generation of thumbnails.

4.5 Gnome-print

Gnome-print is the GNOME printing framework. It uses the Libart imaging model so that applications

can enjoy the same rich imaging model as the canvas' for printing.

In addition, there is a special printer driver for Gnome-print that takes in all the PostScript-like commands and creates canvas primitives for them instead of sending them directly to a printer device. Thus, whatever the application prints is transformed to Bézier paths that appear as items in a canvas. This can be used as a simple "print preview" widget by applications. The canvas allows automatic zooming and scrolling, so applications do not have to be modified at all to support a high-quality print preview.

4.6 Illustration programs

Sodipodi is a powerful illustration program that uses the antialiased canvas for its display. It uses many Libart operations to do computations on Bézier paths.

Gill is a testbed for illustration-related tasks. It parses SVG files and creates the corresponding Bézier, image, text items in the canvas. One of its goals is to support external manipulation of objects via the DOM.

4.7 BEAST

BEAST is a music program that uses BSE, the Be-deviled Sound Engine. It uses the canvas to display filter pipeline graphs and envelope functions for waveforms.

4.8 Gnoghurt

Gnoghurt is a toy program to create video filter pipelines. It uses the canvas to let the user edit these pipelines in an easy way. Gnoghurt provides fruit at the bottom, and must be stirred before eating.

5 Future Work

The canvas has some room for improvement. Here are some of the possible directions in which it may be extended in the future:

- The W3C's Scalable Vector Graphics Specification, or SVG, supports fully hierarchical clipping and opacity adjustment. For example, a whole group of items may be clipped by the result of performing boolean operations on another group of items. Also, the alpha transparency value for a whole group can be changed simultaneously.

The canvas could be extended to support these operations. Groups are already rendered recursively, so controlling the alpha value for a group would be a matter of passing the parent's alpha value to its children. The result would later be composited onto the group's parent's buffer, and so on recursively. Clipping with the result of boolean operations on other groups is more complex, but can be represented in a hierarchical fashion similar to the canvas' current organization.

- Right now the Bonobo component system[3] can be used to proxy canvas items to remote components. This allows for embedding of non-rectangular components in documents. However, this is not as efficient as it could be.

The delayed update/redraw model of the GNOME canvas assumes synchronous updating

and redrawing of individual canvas items, so this is not very efficient for a distributed setting where multiple components may be running on different machines or even on a single multiprocessor machine. The canvas could be extended to support fully asynchronous updates and redraws, asking items to draw themselves in parallel to temporary buffers which would be composited on the fly as they arrive to the parent canvas.

This would allow components running on different processors to draw themselves in parallel and notify the toplevel canvas when they are finished; the canvas would then composite their buffers to form the final result that would be displayed on the screen.

6 Acknowledgments

The GNOME canvas is a collaborative effort. The original version of the canvas was based on the Tk canvas widget. Federico Mena adapted it to the GTK+ object system and extended it with hierarchical groups. Raph Levien wrote the Libart engine and extended the original, GDK-only canvas to support it.

Many people have contributed with ideas and bug fixes to the canvas. The Gnumeric hackers provided excellent bug reports, and coped with our delays in fixing them. Tim Janik pointed out the most bizarre bugs in the antialiased canvas. Owen Taylor provided the scrolling backend for the canvas and was always knowledgeable and helpful with the intricacies of the X window system.

Red Hat, Inc. funded a large part of the development of the canvas, and provided a fun work environment and knowledgeable people. Helix Code, Inc. funded maintenance work of the canvas.

7 Availability

The GNOME canvas is part of the standard `gnome-libs` package, which consists of the core libraries in GNOME. It can be obtained from `ftp.gnome.org` or from the GNOME CVS repository at `cvs.gnome.org`.

Documentation for the canvas is available at <http://developer.gnome.org>.

References

- [1] Adobe Systems, Inc., Bravo Technology Announcement.
- [2] Adobe Systems, Inc., *PostScript Language Reference Manual*, third edition, Addison-Wesley, 1999.
- [3] The Bonobo Component Framework,
<http://developer.gnome.org/arch/-component/bonobo.html>,
<http://developer.gnome.org/doc/-guides/corba/book1.html>.
- [4] The GIMP Drawing Kit (GDK),
<http://www.gtk.org>,
<http://developer.gnome.org/arch/gtk/-gdk.html>.
- [5] The GIMP Toolkit (GTK+),
<http://www.gtk.org>,
<http://developer.gnome.org/arch/gtk>.
- [6] GNU Network Object Model Environment (GNOME), <http://www.gnome.org>.
- [7] Miguel de Icaza, *The Gnumeric Spreadsheet: a Test-Bed for Component Programming*, Proceedings of Linux Expo 1999,
<http://www.gnome.org/gnumeric>.
- [8] Raph Levien, *GtkCanvas and the Next Generation of User Interfaces*, Proceedings of Linux Expo 1999.
- [9] John Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
- [10] Sun Microsystems, Java 2D API,
<http://www.javasoft.com/products/-java-media/2D/index.html>.
- [11] World Wide Web Consortium, Precision Graphics Markup Language Specification (PGML),
<http://www.w3.org/TR/1998/NOTE-PGML>.
- [12] World Wide Web Consortium, Scalable Vector Graphics Specification (SVG),
<http://www.w3.org/Graphics/SVG>.

Efficiently Scheduling X Clients

Keith Packard
XFree86 Core Team, SuSE Inc.
keithp@suse.com

Abstract

The X server is charged with providing window system services to many applications simultaneously, and needs a scheduling mechanism to distribute its limited resources among these applications. The original scheduling mechanism was simplistic and caused graphics-intensive applications to starve interactive applications.

A new scheduling mechanism has been designed which fairly distributes time among the requesting applications while at the same time increasing performance by a small amount. Descriptions of the original and new scheduling mechanism and empirical measurements demonstrating the effects of scheduling within the X server are included along with a discussion on how the design was arrived at.

1 Problems to be solved

1.1 User Feedback Latency

A window system provides an interface between applications and the user, it doesn't receive, transmit or process the underlying data. As such, it focuses on collecting user input, distributing it among the applications and displaying the resulting changes in application state to the screen. For a satisfactory user experience, the delay between event generation and the change in display should be small. As the window system is dependent on the application to generate the appropriate display changes, the delays within the window system should be kept as small as practical.

1.2 Smooth Animation

While the eye will accept more jitter than the ear, animations still require relatively consistent delays between frames. For the window system, this means that when time is scarce, it must parcel it fairly among applications and with fine enough granularity to provide a smooth degradation in frame rates.

1.3 Inter-client synchronization

The X Sync Extension [GCGW91] provides a way of synchronizing X clients with other X clients and various other events. It also provides priorities which are used to schedule among active clients. For this to work well, the X server must minimize the time needed to recognize clients when they become active.

1.4 Speed

Advances in hardware have made the core X graphics operations fast enough in most environments. However, changes which reduce performance are never well received. While X provides for simultaneous active windows, frequently only a single application is drawing. Especially when measuring performance with benchmarks. Any improvements in behavior under heavy load must not negatively impact the common case of a single busy application.

2 Characterizing Scheduler Performance

To effectively analyze the performance of scheduling changes some empirical measurements are helpful.

Based on the problems identified above, the following measurements can be made:

2.1 Measuring Latency

Each X event contains a timestamp marking when the event was generated within the X server. For input events, this should mark the time of the physical action. However, the Linux kernel doesn't provide timestamps marking when the event was received by the kernel, so the X server marks the time they were received by the X server. For mouse events, this is done from within a SIGIO handler and so is relatively close to the physical time.

Applications can cause the X server to generate timestamped PropertyNotify events. By doing this after rendering the feedback and then measuring the difference between the event time and the associated property notify time, a reasonable measurement of user latency can be obtained.

2.2 Animation Update Jitter

Because the X protocol is asynchronous, there can be a large delay from when the application makes a rendering request to when the resulting update appears on the screen. While this delay is important, the jitter generated by differences in scheduling are more visible to the user. Therefore, two different measurements are needed to accurately assess the performance of the system. The delay between when the application generates the rendering requests and when they are displayed measures the latency. The time between displayed frames provides a measure of the jitter in the display.

Again, PropertyNotify events can be used to determine the time at which rendering was completed. The X server computes timestamps for events using the system clock, so as long as the application and X server share the same clock, the delay from generation to display can be measured.

2.3 Measuring Speed

X11perf [MKA⁺94] is an application used to analyze X server performance and to generate benchmark numbers. This tool can be used to accurately

measure the performance impact of changes made to the X server.

3 Existing Practice

The original scheduling algorithm in the X server as developed at Digital in 1987 was relatively simplistic. X applications were modest in their demands on the server as the bulk were simple text-based applications such as emacs or xterm.

Soon thereafter some simple demonstration applications were written which would render the X server essentially unusable. Plaid is one of these. It sends an endless stream of rendering requests to the server of just the right length to tie up the system for long periods of time.

Real applications with such behavior were not far behind, today users have a wide variety of increasingly complex applications that ask much from our venerable protocol. The primitive scheduling which was an occasional annoyance ten years ago has become more of a problem today.

3.1 The Original Algorithm

The sample X server is a single-threaded network server, each client connects to the service using a well known port and sends requests over that connection for processing by the server. Replies, asynchronous events and errors are returned over the same link. As with many single-threaded network server applications, the X server uses `select(2)` to detect clients with pending input.

Once the set of clients with pending input is determined, the X server starts executing requests from the client with the smallest file descriptor. Each client has a buffer which is used to read some data from the network connection, that buffer can be resized to hold unusually large requests, but is typically 4KB. Requests are executed from each client until either the buffer is exhausted of complete requests or after ten requests.

After requests are read from all of the ready clients, the server determines whether any clients still have complete requests in their buffers. If so, the server

foregoes the `select(2)` call and goes back to processing requests for those clients. When all client input buffers are exhausted of complete requests, the X server returns to `select(2)` to await additional data.

3.2 Analysis of this Algorithm

One problem with this algorithm is that it uses a poor metric for the amount of work done by applications. In most cases, ten requests can be executed very rapidly. Applications tend to render only a few objects in each request and the rendering requests are typically quite simple. It is possible, however, to generate requests which run for quite some time, X requests can contain thousands of primitive objects. Ten such requests could take several hundreds of milliseconds to execute. This leads to large variations in the amount of time devoted to each application.

Another issue is that applications which generate few requests are starved for attention. As the X server busily empties the buffers from more active clients, it remains deaf to those with more modest demands. Until every client request buffer is empty of requests, the server doesn't check on the remaining clients.

Less obviously, the server stops processing requests for a client whenever the request buffer doesn't contain a complete request. This means that for rapidly executed large requests, the server calls `select(2)` more often as the request buffer will hold fewer requests.

The benefit of this system is that when clients are busy, the server spends most of its time executing requests and wastes little time on system calls.

4 Available Scheduling Parameters

The information available and the execution context control the design of a scheduler as much as the desired performance characteristics. The X server is a user-mode process written in a high-level language and is designed to be relatively independent of operating system and hardware architecture. This makes the information available limited to that pro-

vided by common OS interfaces and library functions.

4.1 Request Atomicity

The X protocol requires that

the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams). [SG92]

This makes suspending request processing in the middle of a request difficult (in general) as objects manipulated by each request must not appear to other clients in any intermediate state. As a result, the design of the single threaded X server is largely predicated on scheduling clients at request boundaries. There are a few exceptions dealing with network font access which preserve atomicity by changing no global state before suspending request execution.

For the most part, this limitation is not too severe, X requests are usually small and execute quickly. However, when drawing lines and arcs thicker than a single pixel, the server can spend quite some time on a single request. Drawing arcs requires finding three solutions to an eighth order polynomial per scan-line and approximating an elliptical integral for dash lengths. Fortunately, applications largely avoid these primitives as they are so slow making them less of an issue.

Changing this architectural constraint is beyond the scope of this project and so the atomicity requirement translates into a scheduling granularity no finer than a request boundary.

4.2 System Performance

The cost of system calls and other operating-system procedures is important in determining how often each call can be made without a significant impact on overall performance.

Procedure	Time (μ seconds)
Null procedure activation	0.0470
Null syscall (getppid)	0.4708
Gettimeofday	0.8333
Read (1 byte) from /dev/zero	0.9789
Write (1 byte) to /dev/null	0.7709
Select on 10 fd's	9.3765
Signal handler overhead	3.158
UDP latency using localhost	77.3502
TCP latency using localhost	128.1683

Table 1: Performance for Linux System Operations

Measurements made with lmbench [MS96] on a Pentium 300 MHz Mobile MMX laptop by Theodore Ts'o yield the numbers shown in Table 1.

4.3 System Time

In any system which interacts with the user, one reasonable scheduling metric is real time. The X server would check the current time after each request and terminate processing for a client at the appropriate time. However, the current time is only available to user applications through a system call. The X server can execute an "NoOperation" request in 0.360μ seconds, making it more than twice as fast as a call to `gettimeofday(2)`. Calling `gettimeofday(2)` after processing each request would be too expensive, so the X server needs a more efficient mechanism for obtaining the current time.

A low resolution clock can be generated using the interval timer mechanism (`setitimer(2)`). A measure of time can be computed by incrementing a global variable from the signal handler. Signals may occasionally be lost under heavy system load. Under heavy load, the X server would already be starved for CPU resources making scheduling decisions less precise in any case.

The X server should be idle when no work is to be done. When the `ITIMER_REAL` timer is used, `SIGALRM` is delivered even when the server is waiting. As the server suspends using `select(2)`, that system call will immediately return `EINTR` causing the server to execute a significant amount of code before re-entering `select(2)`. To eliminate the load of both a signal handler and another call to `select(2)` on the system, the X server temporarily disables the

timer if it receives a signal while the server is suspended within `select(2)`. When the server awakens again, it restarts the timer.

The X server could use `ITIMER_VIRTUAL` which would report CPU time consumed by the X server. While the X server was idle, no signals would be generated. However, on a busy system, the X server would use progressively larger timer intervals. Additionally, the Linux kernel only accrues CPU time to a process when it happens to be running during the timer interrupt, occasionally increasing the scheduling interval.

4.4 User Intent

Unlike other window systems, X applications register the set of events they are interested in receiving so that unwanted events are not transmitted over the network. When an application receives a requested mouse or keyboard event, the server can infer that the application is likely to generate screen updates based on that event. The scheduler can grant additional resources to those applications.

The X server already used this to a limited extent. Events generated by the user are given priority and are delivered as soon as any current request completes. However, as mouse motion events can be numerous, they were not handled in this manner. Typical modern mouse devices generate fewer than 100 events per second when in motion, not a considerable burden for a modern machine. This behavior was changed to produce the results shown here. Without such changes, pointer motion events would be delayed until a normal scheduling interval. This caused long lags between pointer event generation and application receipt of that event.

4.5 Queued Requests

Another factor available to the scheduler is whether request buffers are empty, contain a partial request or contain one or more complete requests. The existing scheduler uses this to avoid making additional `select(2)` calls. The existence of a partial request makes it likely that the remaining data are waiting in an OS buffer and that a `read(2)` would likely result in additional data.

5 Proposed Scheduler

With the parameters outlined above, the design of the scheduler is straightforward. The goal is to provide relatively fine grained time-based scheduling with increased priority given to applications which receive user input.

5.1 Dynamic Priorities

Each client is assigned a base priority when it connects to the server. Requests are executed from the client with highest priority. Various events cause changes in priority as described below. The Sync extension also applies priorities, the scheduler priorities sort among clients with the same Sync priority so that applications can override the internally generated priorities.

Clients with the same priority are served in round-robin fashion, this keeps even many busy clients all making progress.

5.2 Time Based

Using the `ITIMER_REAL` mechanism outlined above, a copy of the system clock is kept in the X server address space, updated periodically while the X server is active. The resolution of the clock limits the granularity of scheduling and is set to 20ms.

Each client is allowed to execute requests for the same interval after which time other clients are given a turn. If the client is still running at the end of the interval, the client's priority is reduced by one

(but no less than a minimum value). If the client hasn't been ready to run for some time, priority is increased by one (but no more than the initial base value). These priority adjustments penalize busy applications and praise idle ones. This is a simplification of discovering precisely how much time a client has used; that would require a system call.

Each time a client has finished running, the X server recomputes the set of clients ready for execution. That includes examining each client request buffer to determine whether a complete request is already available and also making a call to `select(2)` to discover whether any idle clients have delivered new requests. This is necessary to ensure that requests from higher priority clients are served within a reasonable interval independent of the number of busy low priority clients.

5.3 Monitor User Events

When a client receives user mouse or keyboard events, their priority is raised by one, but no more than a maximum value which is above the base priority. Just as in the Unix scheduler [Tho78], it is desirable to have applications which wait for user input to receive preferential treatment. However, there is no easy way to know whether the application is waiting for input in this case so instead we give modest praise when events are delivered.

5.4 Keep Reading From Clients

When a request buffer no longer contained a complete request, the original scheduler would stop processing a client and wait until `select(2)` indicated that additional data were available. Now `read(2)` is tried first. When that fails to fill the request buffer with a complete request, the server stops processing requests for that client. This increases the number of requests executed before another call to `select(2)` is made.

5.5 Lengthen Timeslice for Single Client

The scheduler monitors how many applications are making requests, when only a single client is making requests for an extended time (one second) the

scheduler increases the amount of time allotted to that client. This improves performance for a single busy application. When another client makes a request, the timeslice interval is returned to normal. While this reduces interactive performance, the effect is transient as it is eliminated as soon as another application is ready to execute.

6 Experimental Setup

The machine used was a Compaq Prosignia with a 466MHz Celeron processor and an S3 Savage4 PCI video board with 32Meg of video ram.

In order to generate reasonable intervals in the animation test, the Linux 2.2.10 kernel was compiled with HZ set to 1000 instead of the default 100. This causes the hardware clock to be programmed for an interval of 1ms instead of the usual 10ms. An alternative would have been to increase the animation interval by a factor of 10, but that would have used an unrealistically low frame rate.

Because the measurements used timestamps reported in X events, they were only accurate to the nearest 1ms. This means that the measurement accuracy of shorter intervals is somewhat limited.

7 Results

To measure the effects of this scheduler, two test cases were written. They were first run on an otherwise idle X server, next they were run while twelve copies of an actively drawing application (plaid) were also running. The idle measurements were (as expected) identical between the two schedulers, only one copy of those results are included in the tables below.

7.1 Interactive Application

The first is a simple interactive feedback demonstration, a rubber band line is drawn following the cursor position. The delay between event generation and receipt is measured (Receipt), along with the total delay from event to the drawing of the line (Echo). See Figure 1.

The old scheduler suffers from significant latency both in delivering events and in scheduling the interactive client for execution. Subjectively, the feedback was jerky and lagged the actual pointer position by long intervals making it difficult to control the application.

The new scheduler also suffers additional latency when other applications are running, but total feedback delay is over 20 times lower (7.4 vs 170) when compared to the old scheduler. There was little perceptible difference between this and the Idle case. Feedback between the mouse and the display was slightly jerky but still quite usable.

7.2 Animation Application

The second test is an animation example. The "xengine" application was modified to delay 15ms between frames. The time between drawing a frame and the subsequent display on the screen was measured (Drawing), along with the time between frames (Frame). See Figure 2.

In this test, the old scheduler suffered from two problems, the long delay between image generation and display (219ms mean), and the large deviation in inter-frame intervals. The long delays would make synchronization with other media, such as audio, difficult. The large range of inter-frame intervals indicates that while busy with other clients, requests from the animation would queue up to be processed several frames at a time. The visible effect was a jerky sequence of images lacking a fluid sense of motion.

Visually, the new scheduler was difficult to distinguish from the idle case. However, the long timeslices granted to each client had measurable effects. The average delay between the generation of the frame and subsequent display is very close to half of the time slice interval. The deviation in frame intervals shows the same effect, that the scheduler was limited in precision by the slice interval.

With both of these tests, the new scheduler is measurably better than the old.

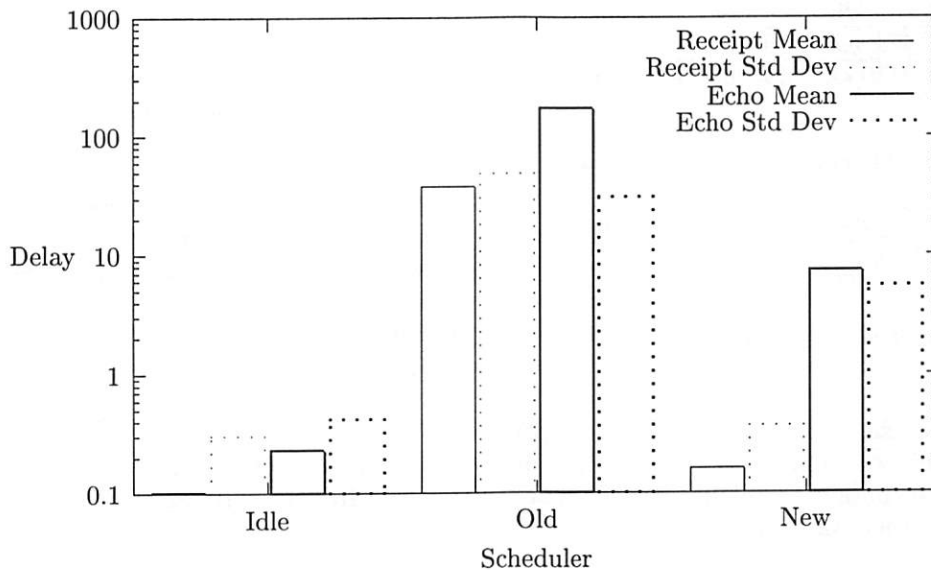


Figure 1: Scheduler Performance for Interactive Test

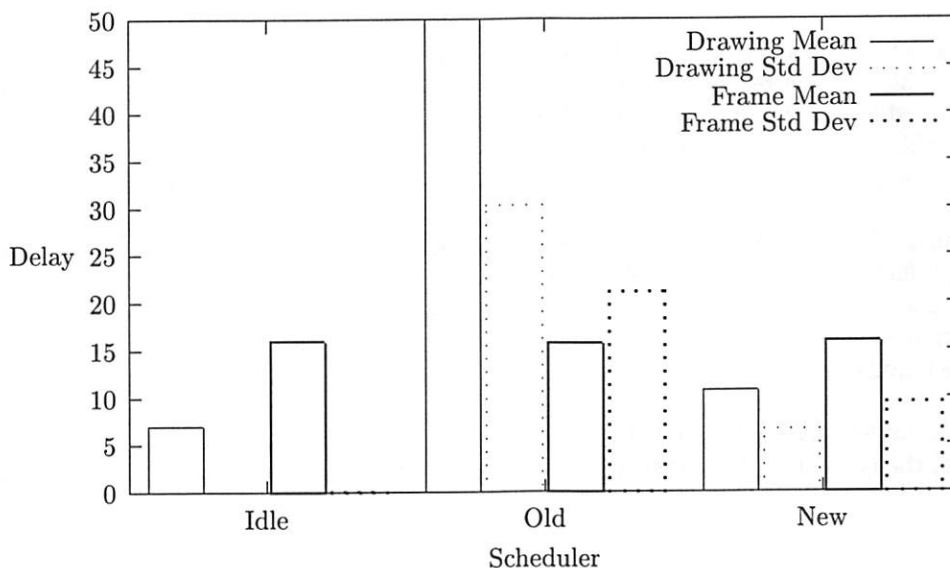


Figure 2: Scheduler Performance for Animation Test

7.3 Performance Measurement

X11perf is a tool used to measure X server performance. It is entirely synthetic, measuring repeated sequences of hundreds of primitive operations. The methods used to collect the data are relatively precise, and with some analysis and understanding, useful data can be extracted.

By inferring what effects can be expected from the scheduler changes, particular x11perf tests can be used to verify those effects. As x11perf is expected to run without competition from other applications, most of the dramatic changes seen above will be absent. One change which seemed promising was the effort to reduce calls to `select(2)`. The old scheduler would call `select(2)` whenever the client request buffer no longer contained a complete re-

Test	Old	New	New/Old
PolyPoint	4740000	4900000	1.03
NoOp	2130000	2680000	1.26

Table 2: X11perf Performance Comparison

	Old	New	New/Old
Xmark	24.9919	25.4732	1.02

Table 3: Xmark Comparison

quest. The new scheduler instead tries a `read(2)` call instead. This will return any pending data or `EWOULDBLOCK` if no data are available, if no data are ready, the server then waits in `select(2)`.

The effect of this change should be greatest for requests which execute quickly, and as seen in table 2 this is indeed the case. The units are operations per second.

Although the tests noted above did show measurable changes, for most of the test results, the two schedulers performed within 2% of each other. This shows that the scheduler changes have little effect on this performance measurement tool.

A standard synthetic benchmark, Xmark, has been created which performs a weighted geometric average of the x11perf test results. While it is more meaningless than most benchmarks, it has, nonetheless gained some popularity.

As the results in table 3 are generated from the x11perf numbers, the result is not surprising.

8 Kernel Timer Granularity

As measured above, the scheduling interval within the X server is not short enough to eliminate significant jitter in animation applications. The scheduling interval within the X server was set to 20ms with the knowledge that the usual Linux kernel timer interrupt was set to 10ms. Thus the operating system limits the ability of the X server to schedule clients precisely.

The 10ms interval has remained unchanged since it was chosen for the BSD Unix implementation for the

Digital VAX machines [LMKQ89]. 100 interrupts per second was considered a negligible load on a machine of that era. With modern machines being somewhat faster, it seems reasonable to consider a higher resolution timer.

Fortunately, Linux provides a single constant which defines the frequency for timer interrupts. The kernel was rebuilt with a 1ms timer interval, and the X server was run with various scheduler intervals to measure the scheduler behavior as well as overall X server performance.

8.1 Interactive Test with Various Scheduler Intervals

For each scheduling interval, the performance of the interactive test was measured. The results are displayed in Figure 3. The mean delay between event generation and receipt is displayed (Receipt) with error bars indicating the standard deviation. The mean delay between event generation and the drawing of the line is also displayed (Echo). Again error bars indicate the standard deviation.

As the X server scheduling interval decreases, the latency between mouse motion and the echo on the screen decreases. Furthermore, the variation in echo latency values also decreases. There is no change in the event receipt latency because the X server checks for pending user input before processing each request.

8.2 Animation Test with Various Scheduler Intervals

For each scheduling interval, the performance of the animation test was measured. The results are displayed in Figure 4. The mean delay between issuing drawing requests and the display on the screen is displayed (Drawing latency) with error bars indicating the standard deviation. The mean time between the display of consecutive frames is displayed (Frame interval) with error bars indicating the standard deviation.

As the X server timer interval is made shorter, the deviation in the frame interval is reduced along with the drawing latency. Reducing the frame interval deviation will make animations appear smoother.

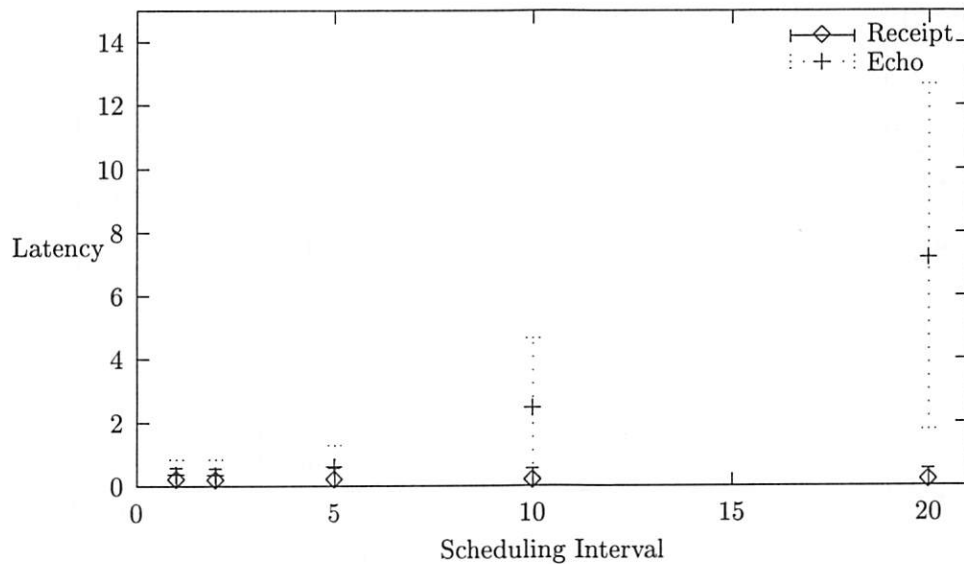


Figure 3: Effects of Changing Scheduler Interval on Interactive Test

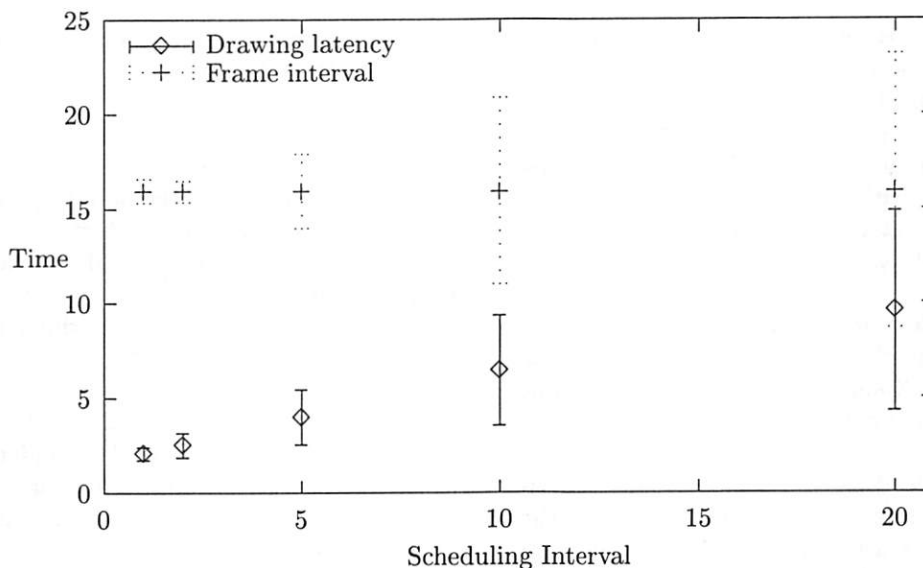


Figure 4: Effects of Changing Scheduler Interval on Animation Test

Reducing the drawing latency will improve synchronization with other media.

8.3 Performance Effects from Various Scheduler Intervals

Two effects should tend to decrease performance with shorter scheduling intervals. As the server uses

a signal to duplicate the kernel clock inside the X server, increasing the frequency of signal delivery will increase the overhead of this process. Further, as the X server checks for other client activity by using `select(2)`, an increase in the scheduling frequency will increase the amount of time spent in `select(2)`.

The new scheduler includes two separate parameters

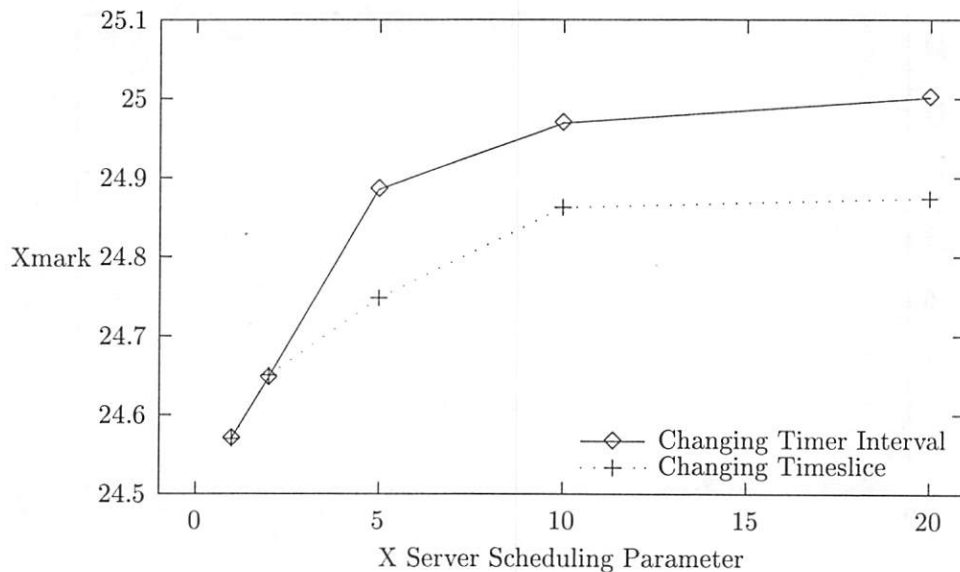


Figure 5: Effects of Changing Scheduler Parameters on Xmark Measurements

to control these two effects, the first is the timer interval and the second is the maximum timeslice granted to the client.

To gauge the overall impact of these two variables, `x11perf` was run with the X server set to use a variety of timer intervals and maximum timeslice values. The results are displayed in Figure 5.

The effect of calling `select(2)` every millisecond is not very large in this X server. There is less than a 2% decrease in Xmarks when decreasing the time between `select(2)` calls from 20ms to 1ms.

While the overall effect is small, particular applications may see larger changes, depending on the balance between X server CPU and graphics accelerator usage. Operations which are limited by the graphics accelerator will show small performance impact, while operations limited by the CPU will show more.

Also evident is that most of the performance impact comes not from a high-resolution timer, but from the cost of `select(2)`. The new scheduling system dynamically increases the timeslice when running a single application, balancing the benefits of reducing system call overhead with providing accurate scheduling for multiple clients.

8.4 Performance Effects of a Kernel Timer Change

The kernel timer is set to a compromise between scheduling precision and the performance impact of additional timer interrupts. Increasing the timer frequency brings an associated increase in kernel processing as it reschedules processes more frequently, possibly context switching more often which affects TLB/cache performance.

`X11perf` was run with identical scheduling parameters with a kernel timer of 1ms and then 10ms, the resulting Xmark numbers appear in Figure 6. As the graph shows, the performance effect of an increase in kernel clock resolution is a decrease of between 1 and 1.5 percent.

8.5 The Kernel Timer Should Change

The measurements above demonstrate that the X server could provide significantly better scheduling if a higher resolution kernel clock were made available. The impact of raising the timer interrupt frequency, at least on X performance, is nominal given the performance of modern hardware.

Graphics hardware is in a unique situation, it provides output without generating interrupts. This

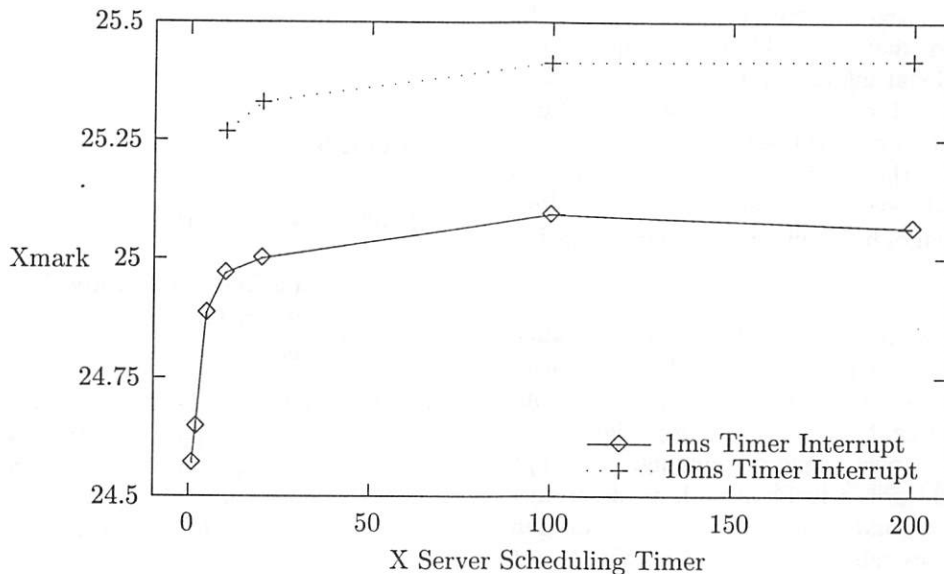


Figure 6: Effects of Changing Kernel Timer on Xmark Measurements

makes it difficult to schedule graphics activities precisely enough to avoid jitter even when the system is idle.

However, the performance impact of such a change needs to be addressed, even the minor 1 to 1.5 percent decrease in performance measured by this change. The measurements made here indicate that the kernel is spending additional cycles managing the increased timer interrupt rate. Perhaps the interaction between the timer interrupt and the upper level timer management code could be adjusted to avoid this overhead.

9 Remaining issues

There are still a few difficult problems which remain unsolved by this new scheduler.

9.1 Threading the X Server

Most X requests are short and execute rapidly, however there are a few core protocol requests and many extension requests which are not so well behaved. A straightforward solution to this problem is to create separate threads executing requests for each

client, and then to build suitable locking mechanisms throughout the server to protect global data. MIT, Data General and Omron cooperated in 1991-1992 to build such a server [Smi92].

The result demonstrated that the rendering engine was a resource that every application needed to access for nearly every request, reducing the multi-threaded X server to a lock-step procession of threads waiting for the display hardware mutex. With multiple screens, a small amount of parallel execution would be possible.

9.2 Multiple Client Performance

The performance optimization of increasing the timeslice granted to a single application to reduce the system call overhead isn't currently done when multiple applications are running. This makes it likely that some requests will execute more slowly under the new scheduler than with the old, but only when more than one client is active.

9.3 Other Kernel Changes

To provide better support for user-mode scheduling, the kernel could make available an inexpensive copy of the system clock. One simple idea would be to

create a shared segment containing a copy of the clock and make that mappable by user-mode programs. Additional information about system load might usefully be included in such a segment. Such a change would improve this scheduler by increasing the resolution of the clock, providing more accurate updates when the system is heavily loaded and eliminate the burden of frequent timer signal generation and reception.

A way of detecting when new data are available for the X server to read without the expense of `select(2)` would be useful. The X server constantly calls `select(2)` to see if any idle clients happen to have new requests pending. Perhaps the server could poll a local variable to determine whether `select` would return different information from the previous call. Such a variable could be set from a signal handler.

10 Conclusion

A simple scheduler, based on what information could be easily obtained by the user-mode X server, demonstrates some significant advantages over the original scheduler without negatively impacting performance. These changes are largely hidden from the user, who will only notice them by the absence of large delays when dealing with applications which flood the X server with requests.

11 Acknowledgments

I thank SuSE for encouraging me to work full time on X. This work was implemented during the X Hot-house event sponsored by SuSE at the Atlanta Linux Showcase in October 1999.

12 Availability

A previous version of this work has been incorporated into the 4.0 release of the X Window System from the XFree86 group. The current version will be available in the next public XFree86 release.

<http://www.xfree86.org>

References

- [GCGW91] Tim Glauert, Dave Carver, Jim Gettys, and David P. Wiggins. X Synchronization Extension Protocol, Version 3.0. X consortium standard, X Version 11 Release 6, 1991.
- [LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Machael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, 1989.
- [MKA⁺94] Joel McCormack, Phil Karlton, Susan Angebrannndt, Chris Kent, Keith Packard, and Graeme Gill. X11perf - x11 server performance test program. Manual page, X11 Version 11 Release 6.4, 1994.
- [MS96] Larry McVoy and Carl Staelin. Im-bench: Portable tools for performance analysis. In *Technical Conference Proceedings*, pages 279–284, San Diego, CA, January 1996. USENIX.
- [SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.
- [Smi92] John Smith. The Multi-Threaded X Server. *The X Resource*, 1:73–89, Winter 1992.
- [Tho78] K. Thompson. Unix implementation. *The Bell System Technical Journal*, 57(6):1931–1946, July-August 1978.

The AT&T AST OpenSource Software Collection

Glenn S. Fowler, David G. Korn, Stephen S. North and Kiem-Phong Vo
AT&T Laboratories – Research
180 Park Avenue, Florham Park, NJ 07932, U.S.A.
gsf,dgk,north,kpv@research.att.com

Abstract

This paper introduces a large collection of reusable software components that AT&T is making available in an OpenSource form. This software has been widely used around the world and includes well-known components such as KornShell, Nmake, Graphviz, Sflo, Vmalloc and Cdt.

1 Introduction

AT&T is not a newcomer to the UNIX market. In fact, it is where UNIX was born. However, AT&T is a newcomer to the world of OpenSource. This paper highlights our entrance into that domain.

UNIX was first invented when AT&T comprised the entire Bell Telephone System including Bell Telephone Laboratories, Western Electric and 23 Regional Operating Companies. This early AT&T company was a regulated monopoly restricted to the telecommunications business and prohibited from entering other businesses. As such, all inventions from Bell Telephone Laboratories unrelated to the telecommunications business could only be licensed externally. The UNIX system fell into this category and was licensed early on for academic use.

In the late '70s and early '80s, AT&T wanted to expand its business into new markets including the computer market. In exchange for this right, AT&T agreed to the now famous consent decree that split off the various Regional Bell Operating Companies or RBOCs. Bell Telephone Laboratories was divided into two parts, AT&T Bell Laboratories and Bellcore (subsequently renamed Telcordia) which performed research and development for the RBOCs. The UNIX system remained with AT&T Bell Laboratories and was commercialized for the first time. With the entry into the computer market, AT&T also became more restrictive in the release of potentially commercializable software.

In the early '90s, AT&T realized that it could not compete effectively in the computer business and began to refocus exclusively into its core strengths, the communications business. As a result, the UNIX

division was sold to Novell. In the subsequent “trivestiture,” the company split off the computer division, NCR, and further separated the communication business into two independent companies, Lucent Technologies for equipment and the current AT&T for services. AT&T Bell Laboratories was divided into two parts, Lucent Bell Laboratories and AT&T Laboratories. All copyrighted software at trivestiture time remained available to both Lucent Bell Laboratories and AT&T Laboratories after the split.

Our software research department, the Advanced Software Technologies department (AST), was formed in the late '70s in the original Bell Telephone Laboratories. It was separate from both UNIX research and development organizations. Our charter was to improve the productivity of AT&T software development. We achieved this by inventing new algorithms and techniques for existing APIs, creating new powerful APIs in the forms of libraries, languages and tools, and developing techniques for accurate build and configuration of software. The audience for our software was the myriad development organizations in AT&T. Since these organizations used diverse equipments and operating systems, a key part of our work was to develop techniques to enhance portability.

Our tools and libraries spread quickly among the AT&T development organizations. To help with internal technology transfer, a separate organization from research was set up to handle support and distribution. Unfortunately, external release of our software was not as smooth. The UNIX development group was responsible for outside release and they were often reluctant to adopt software that they

did not develop. There were, however, notable exceptions such as KornShell, Curses, and Malloc. The latter two libraries eventually became standard parts of Solaris, Irix and Unixware, UNIX variants derived from the System V Release 4.0 UNIX version.

Events turned worse for internal technology transfer after the sale of UNIX to Novell. To reduce the cost for software support and to ensure compliance with development methodologies such as ISO 9000, AT&T development organizations were pushed to buy vendor supported software. In this environment, the most effective avenue to transfer technology from research to development was to license the software to external vendors who, in turn, resold to AT&T.

The recent advance of the OpenSource movement, with the inception of the Open Source Initiative (OSI), opened a new venue for making research software widely accessible. From our point of view, the primary advantages to OpenSource are:

- Increased influence on national and international standardization efforts,
- Creating and supporting alternatives to closed systems,
- The ability to attract vendors to distribute and support the software,
- Improved software quality due to widespread use, and
- Increased visibility of AT&T Laboratories in the research community.

These benefits fit well with the current business directions of AT&T which emphasize on building the communication services business, not selling software. Thus, AT&T intellectual property managers became more open to arguments in favor of an OpenSource software release based on the set of principles called the Open Source Definition as specified by the OSI on the website:

<http://www.opensource.org/osd.html>

In early 1999, we started an effort to release much of the software developed in research under an OpenSource license. After several months of discussions and many rounds of negotiation, AT&T has successfully produced a license that we believe meets the conditions for OpenSource certification and have submitted the license to OSI for official certification.

The AT&T AST OpenSource software represents many years of effort and covers a broad set of libraries and applications. Much of the early work was described in the book Practical Reusable UNIX

Software[2]. The software includes a large subset of the POSIX utilities including the latest version of KornShell. In addition, there are many libraries and utilities not available elsewhere. The software is portable across virtually all UNIX environments, including OpenEdition on MVS (which uses EBCDIC) and Windows systems given a suitable UNIX layer such as UWIN[13].

It is neither possible nor appropriate to describe all the software components here, so the remainder of the paper will focus on our approach to building and packaging software and the terms of our OpenSource license. Interested readers can check the given references for details on particular software components.

2 Libraries

The AT&T AST OpenSource software collection consists of is upward of 3/4 million LOCs. Despite this large size, most of the software was created by a small group of researchers, about 6 at its peak. The software embodies some of the most powerful algorithms and data structures known. For example, the Graphviz package implements our patented graph drawing algorithms that automatically generate pictures of directed and undirected graphs with thousands of nodes and edges in seconds. A focus in our work is to make such algorithms and data structures widely reusable, not just in the tools that we create but also in applications that others would write. Another focus for our software is portability. We wish to make it easy to build applications that transparently compile and execute on at least all different UNIX platforms and certain selected others. The limited human resource and the desire for wide software reuse led us to conclude that:

A major part of our software development effort should be directed toward the creation of powerful reusable libraries that would enable other tools and applications to be built by simply assembling such libraries.

Thus, we embarked on a program to write software libraries that encompass core computing functions such as I/O and memory allocation and other new algorithms and data structures such as data compression and differencing and graph drawing. Below is a partial list of libraries available in the collection:

- *Libast*: This is the porting base library[8] for our software tools. It includes a common header

that provides common data types such as `size_t` and others that may be missing on a particular platform. Similarly, functions are provided to fill in missing ones (e.g., `bcopy()` on non-BSD systems) and to replace existing ones that are inefficient. Libast also provides new convenient functions such as `strperm()` to convert a `chmod` file mode expression into a `mode_t` value.

- *Sfio*: This I/O library[9] provides a robust interface and implements new buffering and data formatting algorithms that are more efficient than those in the standard I/O library, Stdio. For backward compatibility, Sfio also provides emulation code for Stdio that is suitable for both recompiling and relinking Stdio applications.
- *Vmalloc*: This memory allocation library[22] allows creation of different memory regions based on application-defined memory types (heap, shared, memory mapped, etc.) and some library-provided memory management strategies. A backward-compatible Malloc interface is provided that additionally allows an application to selectively perform memory debugging or profiling by setting environment variables.
- *Cdt*: This container data type library[23] provides under a unified interface a comprehensive set of containers: ordered/unordered sets/multisets, lists, stacks and queues. These container data types are based on efficient data structures such as hash tables and splay trees.
- *Libexpr*: This library provides run-time evaluation for simple C-styled expressions. It forms the basis for commands such as `tw`[7], a file tree walker and `cql`[4], a flat file database language.
- *Libgraph*: This graph library[18, 11] supports attributed graphs, generalized nested subgraphs, and stream file I/O in a flexible graph data language. It is built on top of the Cdt library and employs disciplines for I/O, memory management, graph object namespace management and object update callbacks. This library is the base of the Graphviz package to be discussed later.

2.1 Design considerations

Desirable qualities of a reusable library include applicability, efficiency, ease of use and ease of maintenance. However, there is no simple set of rules that can guarantee the simultaneous achievement of these

qualities. The design considerations below serve as guiding principles in our work:

- *Necessity*: This is at the heart of the idea of focusing on writing libraries before applications. In developing tools and applications, we first look into how they can be structured as one or more library functions. In this way, any resulting new APIs would be guaranteed to fill some specific needs and not just arise out of some academic exercise. For example, our versions of the POSIX commands were written first as library functions with the final commands being just simple drivers to parse command line options and make the appropriate function calls. Many of these functions are then reusable as built-ins in other applications such as the shell.
- *Generality*: Each library attempts to encompass as much functionality as possible without sacrificing efficiency. This sometimes means unifying separate but related concepts under a single uniform interface. A good example of this is the memory allocation library, Vmalloc, that unifies under a single interface various memory allocation policies including methods to debug memory errors and to profile memory usage.
- *Consistency*: We make sure that our libraries and functions follow the same interface structures. For example, many libraries provide functions to create structures to store states and return handles referring to such structures. In turn, these handles are used in future function calls to access stored resources. For consistency, we insist that the handle is always the first argument in such calls. This should be contrasted, for example, with Stdio where a `FILE*` handle can be either the first or the last argument.
- *Efficiency*: A reusable library is intended to be used in many applications so performance is a key to its success. This means that its implementation should use the best available algorithms and data structures. However, this is not enough since different applications may have different needs that require specific adaptations. We developed a discipline and method library architecture[24] to unify different algorithms and techniques under a single interface while still allowing applications to tune for efficiency.
- *Modularity*: Modularity is the key in easing maintenance effort as it reduces the interdependency among different libraries as well as among

functions within the same library. To the extent possible, different libraries are kept independent from one another. However, there are exceptions such as memory allocation or I/O that must be negotiated by Vmalloc or Sflo.

- *Irredundancy*: Along the same line of easing maintenance effort, most algorithms and techniques are implemented once in one component, and all other components refer to this one implementation. In this way, we never need to make changes in multiple places for a single algorithmic modification.
- *Extensibility*: Extensibility means the ability to add or change features without breaking existing application code. Most of our libraries are based on the discipline and method architecture[24]. Where applicable, a user-defined discipline structure provides a version field that must be initialized by its caller. Thus, a new library implementation can then use the version field to detect the caller's vintage and act appropriately. The Sflo extended print discipline[10] uses this technique so that a new version can support an obsolete feature for at least one generation before discarding it. In turn, this gives application developers time to adapt their code.
- *Robustness*: Robustness means (1) thoroughly testing functionality, (2) keeping the code free from artificial constraints such as fixed size arrays or integer sizes, and (3) avoiding unsafe interfaces. Dynamic memory allocation is judiciously used to construct any required data structures. An example of the last point is our Sflo function `sfgetr()` that replaces the infamous Stdio `gets()` function and removes any concern about buffer boundary violation. Each of our libraries comes with a comprehensive regression test suite that was built over time based partially on bug reports.
- *Portability*: A basic goal for us is to be able to run our code on all platforms that we have access to. This includes all UNIX/Linux platforms and others such as Windows and MVS. However, portability means more than just that. Our software is configured using the `iffe`[6] approach to target local platform features and maximize performance. For example, the Sflo library selects between memory mapping and other I/O system calls by running a performance test at build time.

3 Tools

Aiming at an efficient, easy to use, and portable computing platform, we have reimplemented nearly all POSIX command tools. We also invented a number of new tools some of which exerted influence on the standards. For example, our KornShell language helped to define the POSIX 1003.2 specification for shell language and our Pax tool for file packaging has been included in POSIX 1003.2. Our tools provide a wide range of functionality. Below are a few examples:

- **nmake**[3]: A far more powerful *make* language that supports dynamic dependency generation and a higher level specification language.
- **iffe**[6]: We use **iffe** to handle architecture-specific features so that such information can be specified in the library source code instead of the makefiles. **iffe** detects architecture-specific features similar **autoconfig**, but at a much more localized level. For example, most GNU packages have a single `config.h`, whereas the Libast library has 25 **iffe** files. By localizing the configuration tests we can limit the amount of code that must be recompiled when individual **iffe** configuration scripts change. Unlike **autoconfig** and **old-make**, **nmake** handles all **iffe** file generation and dependencies automatically as part of the build process.
- **tw**[7]: This command replaces **find** and **xargs** and provides more general searching.
- **cql**[4]: This command provides a C database query language that works on both flat file and binary databases. It performs better than **awk** or **perl**.
- **3d**[5]: This command is a combination of a shell script and a shared library. It modifies the file system semantics to enable viewpathing. That is, different file trees can be virtually overlaid on top of one another with a copy-on-write semantic to the higher layer.
- **warp**: This command is implemented similarly to **3d**. It can be used to run a process as if the time were set to some specified time and the clock speed was set to some specific rate. It was useful for Y2K testing.

The tool development follows a few principles that help to keep the tools uniform, portable, and robust. We discuss these principles next.

- *Conformance to standards:* Our tools conform to the 1992 IEEE POSIX 1003.2 standard[20] and provide additional extensions where appropriate. Thus, at the least, strictly conforming POSIX applications can use our code without change. We have rewritten most of the tools in the base standard and User Portability sections except for a few such as **awk**, **diff**, **mailx**. We imported the GNU version of **diff** and the BSD version of **mailx** and enhanced the latter to add features such as MIME enclosures and IMAP.
- *Conformance to common conventions:* On various platforms, many common utilities provide extensions that are beyond the standard but well liked by users. Whenever possible, our versions of the utilities provide the same extensions. For example, most of the utilities accept the long name options that GNU supports. To handle conflicts between extensions done in different universes such as BSD or System V, we added a configuration option **UNIVERSE** to control the behavior of such utilities. In addition, the **getconf** utility is extended so that users can set configuration parameters such as **UNIVERSE**.
- *Avoiding absolute pathnames:* One of the banes of binary distributions is embedded path names. For example, the **file** command may consult **/etc/magic** for file descriptions. Its not appropriate for the our **file** command to look in the same place. Instead of adding a new environment variable for each candidate fixed path, we use the **\$PATH** environment variable, which must be set properly to use any package, to locate command related files. The convention is: the files for command *foo* are found by doing a **\$PATH** search for the **../lib/foo** directory. A binary directory tree can be installed anywhere without recompilation; the only requirement is that the binary **bin** directory is added to **\$PATH**.
- *Avoiding size restrictions:* Our tools are freed artificial constraints such as fixed size arrays or small file sizes. For example, even though the POSIX standard only requires handling of text files with line lengths 2K or less, our tools do not have such line length limits. Instead, they use the record-reading function **sfgetr()** in the **Sfio** library which allocates dynamic memory as necessary to buffer lines with arbitrary lengths. Via the use of **Sfio**, our tools also transparently work on systems that support files with offsets larger than 32 bits.
- *Combining related utilities into one:* Whenever multiple related utilities can be combined into one, we do so. For example, our **pax** command combines the functionality of **cpio** and **tar**. In addition, it supports multiple formats such as ANSI and EBCDIC standard labelled tape, VMS backup and Microsoft cabinet files and also compression methods such as **compress**, **gzip**, and our own **Vdelta**[12] method for compression and differencing. Other notable examples include the unification of **grep**, **fgrep**, and **egrep** in a single command, the support of MIME Base64 and Binhex encodings in **uuencode** and **uudecode**, and the support of MD5 hashing in **cksum**.
- *Self-documenting tools:* Our tools use a common library function **optget()** to parse command line arguments. We extended this option parser so that it can generate the man page in one or more formats. For example, **pax --man**, **pax --nroff**, and **pax --html** would generate the documentation on the screen, in **Troff**, or in **HTML** format.
- *Building and using powerful reusable libraries:* It is worth emphasizing again here that our main focus is on building powerful reusable libraries that can be easily assembled into commands. This helps to maximize the reusability of the code. For example, even the KornShell itself was implemented as a library. In turn, this enabled the creation of **tksh**[14], a combination of shell and the Tk graphics library[19].

4 Graph visualization

A large part of our work is in software reengineering and data visualization. To help with this effort, we developed **Graphviz**[11], a collection of portable tools for rendering and interacting with abstract graph (network) drawings.

The main **Graphviz** layout programs, **dot** and **neato**, read text specifications of the nodes and edges of a graph, and emit drawings in a graphics language such as Postscript, **pic**, **GIF**, **Metapost**, **VRML** or **PNG**. We have combined these rendering engines with the graphics editor **lefty**[15] to build an interactive diagram manipulator **dotty**[16]. To provide support for a more popular scripting environment, **Graphviz** can be compiled as a **Tcl/Tk** extension.

Graphviz was also made into a web server by adding **GIF** and **PNG** drivers and a wrapper script

to run **dot** or **neato** as a remote cgi-bin service. This service is now employed within AT&T and by outside projects (for example, mozilla.org) and represents a simple experiment in how programs providing lightweight services can be reused more easily as web services than as packages that must be manually downloaded, installed and kept up-to-date. Also, for more sophisticated user interface customization we created Grappa, a Java graph library that communicates with **dot** as a server in the same way as **lefty/dotty**. Grappa can run standalone or as an applet.

Recent work on Graphviz addressed dynamic layout, where diagrams are maintained on-line with stable incremental updates. This work required devising event-based APIs and modifying the front ends to handle layout event streams instead of batch layout. The OpenSource code release includes a component addressing the Microsoft Windows platform. Our goal here was to create a fully OLE-aware network diagram editor. The editor is embeddable, may contain foreign objects as content and can be controlled via C++ or scripting languages such as Visual Basic. To achieve this, we factored the diagram editor into a generic OLE client-server, Montage[25], that provides UI management, persistence for non-hierarchical collections of objects, and domain-specific components to interpret user interface events or manage layouts. This facet of the Graphviz project diverged a great deal from our UNIX roots. However, Montage is a valuable contribution to the OpenSource community. To our knowledge it is the only fully OLE-aware software component available in an OpenSource form, and perhaps the only one outside of Microsoft, Visio, and Borland/Inprise. Moreover, the container library is a separate, clean design well suited to reuse, not laden with application-specific semantics.

5 Packaging

We employ a packaging process designed to ease the reproduction of our environment on multiple platforms. This process provides a mechanism for bundling a set of source components, transporting them to another platform, making binaries from source using the native compilation system, and bundling binaries for use on other equivalent platforms.

5.1 Packages

The smallest distribution unit is a *package*, i.e., a collection of source or binary *components*. A *source*

component is a group of source files controlled by a *makefile*; the makefile and all related source reside in a directory named by the component. *Making* a component generates binaries from the source and makes them available for use. A *binary component* contains all of the generated binaries corresponding to a source component. Most components generate commands or libraries and interface files, but some may provide only documentation or data.

A component may depend on other components. These dependencies define a *make order*. That is, a component may require that other components be made before it is made. Packages might also have dependencies. For example, the Graphviz package requires the Libast package.

The traditional method to handle component make order to hard-wire the order in package makefiles or build scripts, with the restriction that each package reside in its own directory hierarchy. This method defeats any attempts at sharing code between packages. For example, the GNU *fileutils*, *findutils*, and *textutils* packages each have a *lib* source directory. Out of a combined 152 files, 53 are unique to one package, 30 are shared between two, and 13 are shared between all three. This is an unacceptable situation and inevitably leads to duplication and splintering – maintenance nightmares for a small group like ours.

nmake solves the component make order problem. Given a collection of component makefiles, **nmake** constructs a component dependency graph, and makes the components in order, using a separate **nmake** invocation for each component. Independent components are detected to facilitate concurrent makes. It is important to note that when components are added to a package hierarchy the new components are automatically detected and made in the proper order. This means that, barring generated binary name clashes, packages and components can be freely and safely combined.

5.2 Versioning

As mentioned, there are two types of packages: source and binary. These come in two flavors: *base* and *delta*. A *base* package contains a complete copy of all the package components. A *delta* package is similar to a *patch*: it contains only the differences from its base package. Unlike a patch, delta differences are maintained at a byte level instead of a text line-by-line level. This allows binary deltas as well as source deltas. Delta packages also contain *delete* information, so that files may be deleted from a component as it ages. Packages are simply compressed

archive files maintained by our **pax** command which computes deltas using our **vdelta** algorithm. Support files are simply added to the archive of component source as the packages are generated.

Delta packages form the basis of a simple but complete version management strategy. With a little discipline we are able to record and document software updates and bug fixes. Each component has a **RELEASE**, **CHANGES**, or **CHANGELOG** file that contains a dated comment line for each notable change, in reverse chronological order (newest entries at the top.) After component source is modified and tested (including **RELEASE** file edits), a delta source package is made to record all of the changes. Delta source packages are quite efficient in space and typically take up less than 1 percent of the corresponding base source packages.

Each package is stamped with its creation date and delta count (starting at 1). As a package archive is written, its stamp and the stamps of packages it depends on are recorded as files in the archive. When unpacking a package a simple sort on the stamp files tells if the proper dependent packages are present. Since backwards compatibility is guaranteed, the stamp checking rule is simple: any package stamp equal to or newer than the requested stamp is acceptable.

5.3 Source vs. Binary

The package layout maintains source files (readonly) in a separate directory tree from binary files (generated). This feature allows many binary packages to be made from a single source copy, handy when the package directory tree is cross-mounted on hosts with different architectures. Files in the binary directory tree take precedence over files in the source tree. So local modifications can be simply done by copying source files to the binary tree and modifying them there. In this way, the original and modified files can be compared, and changes made for one architecture won't interfere with the others. A source delta (patch) is simply made by recording the differences between source files in the binary and source trees.

5.4 The package Command

The **package** command is the interface for all package management. This command is part of the **INIT** package that all other packages require. The **INIT** package must first be downloaded into an empty directory tree and installed (**gunzip < INIT-yyyy-mm-dd | tar xvf -**). Then other packages

can then be downloaded, made, and/or installed by the **package** command.

Each package has a description file, *package.pkg* (an **nmake** makefile), that lists its components and package dependencies. Any component described by an **nmake** makefile can be part of a package; no other files or auxiliary package information is required. The package is used for the following operations:

- **write [base|delta] [binary|source] package**: This creates an archive for *package*, including version stamp and binary checksum support files.
- **read [file.tgz|file.nnn]**: This reads the package base archive *file.tgz* or package delta archive *file.nnn*.
- **make [package]**: This makes and installs the binaries for *package*, or all packages if *package* is omitted.
- **verify [package]**: This verifies the installed binaries for *package*, or all packages if *package* is omitted, against the package checksum files.
- **test [package]**: This runs the regression tests for *package*, or all packages if *package* is omitted.
- **use [uid|package]**: This runs an interactive shell with environment initialized for using *package* or the package installed by the user *uid*. An unfortunate side effect of using shared libraries (DLLs) is that some systems require specific (and different for every system) environment variable settings to properly locate the DLLs at runtime.

6 License terms

The AT&T Source Code agreement was written both to satisfy the Open Source Definition, and to protect AT&T's intellectual property and other rights. Before creating a new license, we carefully reviewed the main licenses already in use, particularly, GPL, LGPL, QPL, Apple Public Source License, and the IBM open source license. These were not satisfactory to AT&T as, for example, they do not adequately cover patent rights. Often such holes are just as detrimental to licensees as they are to the licensor.

The AT&T Source Code Agreement (ASCA 1.2D), listed in the Appendix, gives licensees the right to:

- Read, study, display, compile, and execute binaries made from the source code.
- Use AT&T patents in the original code to execute original or modified software.
- Redistribute the original source package in any format, as long the contents are preserved exactly.
- Distribute patches that include the copyrighted source code.
- Distribute binaries made from original or modified source code.

From a licensee's standpoint, the main conditions are:

- Distribution can only be made to those that agree to the license terms,
- If modifications to the source are made public, then AT&T has the right to include these changes in its package, and
- AT&T controls the contents of the official source distribution.

In many other ways the license terms are liberal toward commercial OpenSource licensees. The ASCA allows licensees to charge for redistribution. It also grants the right to use AT&T patents involved in the code, even when modified.

Some of the concerns brought up in the OpenSource review include those below. We felt that these license terms were reasonable, and left them in.

- A restriction against framing the AT&T website, perhaps to suggest a relationship with AT&T.
- Your rights under this agreement can be terminated automatically if we receive a non-frivolous claim of a patent infringement by a third party related to the source code on our website. In the event of an infringement claim, you would have to replace any infringing portion of the source code with non-infringing code or license the patent from the third party. For this reason, the agreement requires you to periodically check the AT&T website for such infringement notices.
- A "no strict construction" clause. This is an agreement that the license should not somehow be construed literally against what was really intended, to the detriment of either party.

Some very significant improvements were made as part of the OpenSource negotiation, for example, plugging holes in the license concerning the equivalent of "fair use" (since the ASCA relies on granting specific rights, instead of transferring ownership of a copy of a copyrighted work) and avoiding cumbersome restrictions on charging for redistributed copies, shrink-wrap licensing and repackaging of the source, among others. The AT&T OpenSource license agreement can also be viewed at:

<http://www.research.att.com/sw/license/ast-open.html>

7 Conclusion

This paper introduced the AT&T AST OpenSource software collection. This software collection is nearly twenty years in the making and includes tools such as the KornShell language, the Nmake system, the Graphviz package for graph drawing, and core computing and algorithm libraries such as Sflo, Cdt and Vmalloc. Many of the components were previously available for non-commercial use from the website:

<http://www.research.att.com/sw/tools/>

In just the past year and a half, more than 40,000 copies have been downloaded. Many large and mission-critical projects both within AT&T as well as around the world are dependent on these components. A frequently asked question from external users is "How do we license the software for production use?" This OpenSource release provides an answer.

Acknowledgments

The authors thank Ben Lee and Tom Restaino for legal work, and Dave Belanger, Jeff George and David Nagel for backing the Open Source release.

References

- [1] ANSI. *American National Standard for Information Systems - Programming Language - C*. American National Standards Institute, 1990.
- [2] Edited by B. Krisnamurthy. *Practical Reusable Unix Software*. John Wiley & Sons, Inc., 1995.
- [3] Glenn S. Fowler. The Fourth Generation Make. In *Proc. of the USENIX 1985 Summer Conference*, pages 159-174, 1985.

- [4] Glenn S. Fowler. cql – A Flat File Database Query Language. In *Proc. of the USENIX Winter 1994 Conference*, pages 11–21, January 1994.
- [5] Glenn S. Fowler, David G. Korn, and Herman C. Rao. n-DFS: The Multiple Dimensional File System. *Trends in Software: Configuration Management*, 2, 1994.
- [6] Glenn S. Fowler, David G. Korn, J.J. Snyder, and Kiem-Phong Vo. Feature-Based Portability. In *Proc. of the Usenix VHLL Conference*, pages 197–207. USENIX, 1994.
- [7] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. An Efficient File Hierarchy Walker. In *Proc. of the Summer '89 Usenix Conference*, pages 173–188. USENIX, 1989.
- [8] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for Writing Reusable Library. In *Proc. of the ACM SIGSOFT Symposium on Software Reusability*, pages 150–160. ACM Press, 1995.
- [9] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Sflo: A Buffered I/O Library. *Software—Practice and Experience*, Accepted for publication, 1999.
- [10] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Extended Data Formatting Using Sflo. In *Proc. of the 2000 Usenix Conference*, USENIX, 2000.
- [11] E. R. Gansner, E. Koutsofios, S. C. North, and Kiem-Phong Vo. Graph Visualization in Software Analysis. In *Proc. of the Symp. on Assessment of Quality Software Development Tools*, pages 226–237, 1992.
- [12] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Software Configuration and Maintenance Workshop*, 1996.
- [13] David G. Korn. Porting UNIX to Windows NT. In *Proc. of the 1997 Usenix Conference*. USENIX, 1997.
- [14] Jeffrey L. Korn. Tksh: A Tcl library for KornShell. In *Proc. of the USENIX Tcl/Tk Workshop*, pages 149–159, Monterey, CA, July 1996.
- [15] Eleftherios Koutsofios and David Dobkin. Lefty: A two-view editor for technical pictures. In *Proc. of Graphics Interface '91*, pages 68–76, 1991.
- [16] Eleftherios Koutsofios and Stephen C. North. Applications of Graph Visualization. In *Proc. of Graphics Interface 1994 Conference*, pages 235–245, Banff, Canada, May 1994.
- [17] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1995.
- [18] Stephen C. North and Kiem-Phong Vo. Dictionary and Graph Libraries. In *Proc. of the Winter '93 Usenix Conference*, pages 1–11. USENIX, 1993.
- [19] John K. Ousterhout. *Tcl and the Tk Toolkit*. Reading, MA, 1994.
- [20] POSIX. *IEEE Standard 1003.2-1992, ISO/IEC 9945-2:1992, POSIX – Part 2: Shell and Utilities*. Institute of Electrical and Electronics Engineers, 1993.
- [21] Robert Sedgewick. *Algorithms, 2nd Edition*. Addison-Wesley, 1988.
- [22] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software—Practice and Experience*, 26:1–18, 1996.
- [23] Kiem-Phong Vo. Cdt: A Container Data Type Library. *Software—Practice and Experience*, 27:1177–1197, 1997.
- [24] Kiem-Phong Vo. An Architecture for Reusable Libraries. In *Proc. of the 5th International Conference on Software Reuse*. IEEE, 1998.
- [25] Gordon Woodhull and Stephen C. North. Montage – an ActiveX Container for Dynamic Interfaces. In *Proc. of the 2nd USENIX Windows NT Symposium*, pages 109–116, Seattle, CA, August 1998.

AT&T AST OpenSource SOURCE CODE AGREEMENT

Version 1.2D

PLEASE READ THIS AGREEMENT CAREFULLY. By accessing and using the **Source Code**, you accept this Agreement in its entirety and agree to only use the **Source Code** in accordance with the following terms and conditions. If you do not wish to be bound by these terms and conditions, do not access or use the **Source Code**.

1. YOUR REPRESENTATIONS

1. You represent and warrant that:

- a. If you are an entity, or an individual other than the person accepting this Agreement, the person accepting this Agreement on your behalf is your legally authorized representative, duly authorized to accept agreements of this type on your behalf and obligate you to comply with its provisions;
- b. You have read and fully understand this Agreement in its entirety;
- c. Your **Build Materials** are either original or do not include any **Software** obtained under a license that conflicts with the obligations contained in this Agreement;
- d. To the best of your knowledge, your **Build Materials** do not infringe or misappropriate the rights of any person or entity; and,
- e. You will regularly monitor the **Website** for any notices.

2. DEFINITIONS AND INTERPRETATION

1. For purposes of this Agreement, certain terms have been defined below and elsewhere in this Agreement to encompass meanings that may differ from, or be in addition to, the normal connotation of the defined word.
 - a. "**Additional Code**" means **Software** in source code form which does not contain any
 - i. of the **Source Code**, or
 - ii. derivative work (such term having the same meaning in this Agreement as under U.S. Copyright Law) of the **Source Code**.
 - b. "**AT&T Patent Claims**" means those claims of patents (i) owned by AT&T and (ii) licensable without restriction or obligation, which, absent a license, are necessarily and unavoidably infringed by the use of the functionality of the **Source Code**.
 - c. "**Build Materials**" means, with reference to a **Derived Product**, the **Patch** and **Additional Code**, if any, used in the preparation of such **Derived Product**, together with written instructions that describe, in reasonable detail, such preparation.
 - d. "**Derived Product**" means a **Software Product** which is a derivative work of the **Source Code**.
 - e. "**IPR**" means all rights protectable under intellectual property law anywhere throughout the world, including rights protectable under patent, copyright and trade secret laws, but not trademark rights.
 - f. "**Package**" means a computer file containing the exact same contents as the computer file from the **Website** which will be downloaded after accepting, or was opened to access, this Agreement.
 - g. "**Patch**" means **Software** for changing all or any portion of the **Source Code**.
 - h. "**Proprietary Notice**" means the following statement:

This product contains certain software code or other information ("AT&T Software") proprietary to AT&T Corp. ("AT&T"). The AT&T Software is provided to you "AS IS". YOU ASSUME TOTAL RESPONSIBILITY AND RISK FOR USE OF THE AT&T SOFTWARE. AT&T DOES NOT MAKE, AND EXPRESSLY DISCLAIMS, ANY EXPRESS OR IMPLIED

WARRANTIES OF ANY KIND WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, WARRANTIES OF TITLE OR NON-INFRINGEMENT OF ANY INTELLECTUAL PROPERTY RIGHTS, ANY WARRANTIES ARISING BY USAGE OF TRADE, COURSE OF DEALING OR COURSE OF PERFORMANCE, OR ANY WARRANTY THAT THE AT&T SOFTWARE IS "ERROR FREE" OR WILL MEET YOUR REQUIREMENTS. Unless you accept a license to use the AT&T Software, you shall not reverse compile, disassemble or otherwise reverse engineer this product to ascertain the source code for any AT&T Software.

© AT&T Corp. All rights reserved. AT&T is a registered trademark of AT&T Corp.

- i. "**Software**" means, as the context may require, source or object code instructions for controlling the operation of a central processing unit or computer, and computer files containing data or text.
 - j. "**Software Product**" means a collection of computer files containing **Software** in object code form only, which, taken together, reasonably comprise a product, regardless of whether such product is intended for internal use or commercial exploitation. A single computer file can comprise a **Software Product**.
 - k. "**Source Code**" means the **Software** contained in compressed form in the **Package**.
 - l. "**Website**" means the Internet website having the URL <http://www.research.att.com/sw/download/> AT&T may change the content or URL of the **Website**, or remove it from the Internet altogether.
2. By way of clarification only, the terms **Package**, **Proprietary Notice** and **Source Code** when used in this Agreement shall mean the materials and information defined by such terms without any change, enhancement, amendment, alteration or modification (collectively, "changes").

3. GRANT OF RIGHTS

1. Subject to third party intellectual property claims, if any, and the terms and conditions of this Agreement, AT&T grants to you under:
 - a. the **AT&T Patent Claims** and AT&T's copyright rights in the **Source Code**, a non-exclusive, fully paid-up license to:
 - i. Reproduce and distribute the **Package**;
 - ii. Display, perform, use, and compile the **Source Code** and execute the resultant binary **Software** on a computer;
 - iii. Prepare a **Derived Product** solely by compiling **Additional Code**, if any, together with the code resulting from operating a **Patch** on the **Source Code**; and,
 - iv. Execute on a computer and distribute to others **Derived Products**, except that, with respect to the **AT&T Patent Claims**, the license rights granted in clauses (iii) and (iv) above shall only extend, and be limited, to that portion of a **Derived Product** which is **Software** compiled from some portion of the **Source Code**; and,
 - b. AT&T's copyright rights in the **Source Code**, a non-exclusive, fully paid-up license to prepare and distribute **Patches** for the **Source Code**.
2. Subject to the terms and conditions of this Agreement, you may create a hyperlink between an Internet website owned and controlled by you and the **Website**, which hyperlink describes in a fair and good faith manner where the **Package** and **Source Code** may be obtained, provided that, you do not frame the **Website** or otherwise give the false impression that AT&T is somehow associated with, or otherwise endorses or sponsors your website. Any goodwill associated with such hyperlink shall inure to the sole benefit of AT&T. Other than the creation of such hyperlink, nothing in this Agreement shall be construed as conferring upon you any right to use any reference to AT&T, its trade names, trademarks, service marks or any other indicia of origin owned by AT&T, or to indicate that your products or services are in any way sponsored, approved or endorsed by, or affiliated with, AT&T.

3. Except as expressly set forth in Section 3.1 above, no other rights or licenses under any of AT&T's **IPR** are granted or, by implication, estoppel or otherwise, conferred. By way of example only, no rights or licenses under any of AT&T's patents are granted or, by implication, estoppel or otherwise, conferred with respect to any portion of a **Derived Product** which is *not* **Software** compiled from some portion, without change, of the **Source Code**.

4. YOUR OBLIGATIONS

1. If you distribute **Build Materials** (including if you are required to do so pursuant to this Agreement), you shall ensure that the recipient enters into and duly accepts an agreement with you which includes the minimum terms set forth in Appendix A, *see the website for this Appendix*, (completed to indicate you as the **LICENSOR**) and no other provisions which, in AT&T's opinion, conflict with your obligations under, or the intent of, this Agreement. The agreement required under this Section 4.1 may be in electronic form and may be distributed with the **Build Materials** in a form such that the recipient accepts the agreement by using or installing the **Build Materials**. If any **Additional Code** contained in your **Build Materials** includes **Software** you obtained under license, the agreement shall also include complete details concerning the license and any restrictions or obligations associated with such **Software**.
2. If you prepare a **Patch** which you distribute to anyone else you shall:
 - a. Contact AT&T, as may be provided on the **Website** or in a text file included with the **Source Code**, and describe for AT&T such **Patch** and provide AT&T with a copy of such **Patch** as directed by AT&T; or,
 - b. Where you make your **Patch** generally available on your Internet website, you shall provide AT&T with the URL of your website and hereby grant to AT&T a non-exclusive, fully-paid up right to create a hyperlink between your website and a page associated with the **Website**.
3. If you prepare a **Derived Product**, such product shall conspicuously display to users, and any corresponding documentation and license agreement shall include as a provision, the **Proprietary Notice**.

5. YOUR GRANT OF RIGHTS TO AT&T

1. You grant to AT&T under any **IPR** owned or licensable by you which in any way relates to your **Patches**, a non-exclusive, perpetual, worldwide, fully paid-up, unrestricted, irrevocable license, along with the right to sublicense others, to (a) make, have made, use, offer to sell, sell and import any products, services or any combination of products or services, and (b) reproduce, distribute, prepare derivative works based on, perform, display and transmit your **Patches** in any media whether now known or in the future developed.

6. AS IS CLAUSE / LIMITATION OF LIABILITY

1. The **Source Code** and **Package** are provided to you "AS IS". YOU ASSUME TOTAL RESPONSIBILITY AND RISK FOR YOUR USE OF THEM INCLUDING THE RISK OF ANY DEFECTS OR INACCURACIES THEREIN. AT&T DOES NOT MAKE, AND EXPRESSLY DISCLAIMS, ANY EXPRESS OR IMPLIED WARRANTIES OF ANY KIND WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, WARRANTIES OF TITLE OR NON-INFRINGEMENT OF ANY **IPR** OR TRADEMARK RIGHTS, ANY WARRANTIES ARISING BY USAGE OF TRADE, COURSE OF DEALING OR COURSE OF PERFORMANCE, OR ANY WARRANTY THAT THE **SOURCE CODE** OR **PACKAGE** ARE "ERROR FREE" OR WILL MEET YOUR REQUIREMENTS.
2. IN NO EVENT SHALL AT&T BE LIABLE FOR (a) ANY INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, LOSS OF PROGRAMS OR INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE THE **SOURCE CODE** OR **PACKAGE**, EVEN IF AT&T OR ANY OF ITS AUTHORIZED REPRESENTATIVES

HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, (b) ANY CLAIM ATTRIBUTABLE TO ERRORS, OMISSIONS, OR OTHER INACCURACIES IN THE **SOURCE CODE OR PACKAGE**, OR (c) ANY CLAIM BY ANY THIRD PARTY.

3. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IN THE EVENT THAT APPLICABLE LAW DOES NOT ALLOW THE COMPLETE EXCLUSION OR LIMITATION OF LIABILITY OF CLAIMS AND DAMAGES AS SET FORTH IN THIS AGREEMENT, AT&T'S LIABILITY IS LIMITED TO THE GREATEST EXTENT PERMITTED BY LAW.

7. INDEMNIFICATION

1. You shall indemnify and hold harmless AT&T, its affiliates and authorized representatives against any claims, suits or proceedings asserted or commenced by any third party and arising out of, or relating to, your use of the **Source Code**. This obligation shall include indemnifying against all damages, losses, costs and expenses (including attorneys' fees) incurred by AT&T, its affiliates and authorized representatives as a result of any such claims, suits or proceedings, including any costs or expenses incurred in defending against any such claims, suits, or proceedings.

8. GENERAL

1. You shall not assert against AT&T, its affiliates or authorized representatives any claim for infringement or misappropriation of any **IPR** or trademark rights in any way relating to the **Source Code**, including any such claims relating to any **Patches**.
2. In the event that any provision of this Agreement is deemed illegal or unenforceable, AT&T may, but is not obligated to, post on the **Website** a new version of this Agreement which, in AT&T's opinion, reasonably preserves the intent of this Agreement.
3. Your rights and license (but not any of your obligations) under this Agreement shall terminate automatically in the event that (a) notice of a non-frivolous claim by a third party relating to the **Source Code** or **Package** is posted on the **Website**, (b) you have knowledge of any such claim, (c) any of your representations or warranties in Article 1.0 or Section 8.4 are false or inaccurate, (d) you exceed the rights and license granted to you or (e) you fail to fully comply with any provision of this Agreement. Nothing in this provision shall be construed to restrict you, at your option and subject to applicable law, from replacing the portion of the **Source Code** that is the subject of a claim by a third party with non-infringing code or from independently negotiating for necessary rights from the third party.
4. You acknowledge that the **Source Code** and **Package** may be subject to U.S. export laws and regulations, and, accordingly, you hereby assure AT&T that you will not, directly or indirectly, violate any applicable U.S. laws and regulations.
5. Without limiting any of AT&T's rights under this Agreement or at law or in equity, or otherwise expanding the scope of the license and rights granted hereunder, if you fail to perform any of your obligations under this Agreement with respect to any of your **Patches** or **Derived Products**, or if you do any act which exceeds the scope of the license and rights granted herein, then such **Patches**, **Derived Products** and acts are not licensed or otherwise authorized under this Agreement and such failure shall also be deemed a breach of this Agreement. In addition to all other relief available to it for any breach of your obligations under this Agreement, AT&T shall be entitled to an injunction requiring you to perform such obligations.
6. This Agreement shall be governed by and construed in accordance with the laws of the State of New York, USA, without regard to its conflicts of law rules. This Agreement shall be fairly interpreted in accordance with its terms and without any strict construction in favor of or against either AT&T or you. Any suit or proceeding you bring relating to this Agreement shall be brought and prosecuted only in New York, New York, USA.

Implementing Internet Key Exchange (IKE)

Niklas Hallqvist
Applitron Datasystem AB
niklas@openbsd.org

Angelos D. Keromytis
Distributed Systems Lab, University of Pennsylvania
angelos@openbsd.org

Abstract

A key component of the IP Security architecture is the Internet Key Exchange protocol. IKE is invoked to establish session keys (and associated cryptographic and networking configuration) between two hosts across the network. IKE needs to authenticate and authorize the parties involved in an exchange, negotiate parameters to be used for the communication, and interact with the local IPsec stack. The number of tasks, along with the flexibility built into the protocol, as well as the need to allow future additions and modifications to the protocol, need to be taken into consideration when designing and implementing IKE.

Another complicating factor is the need for security policy management. Although IKE can establish security associations with remote hosts, some method for determining what kinds of traffic can and should be exchanged with a remote host is necessary. As there is no standard specification yet, we are using a trust-management based approach using the KeyNote system as a basis for specifying policy.

This paper discusses the design, architecture, and implementation details of the OpenBSD IKE daemon, with separate mention of the security policy mechanism.

1 Introduction

The IP Security architecture [14], as specified by the IETF (Internet Engineering Task Force), is comprised of a set of protocols that provide data integrity, confidentiality, replay protection, and au-

thentication at the network layer. This positioning in the network stack offers considerable flexibility in transparently employing IPsec in different roles (*e.g.*, in building Virtual Private Networks, end-to-end security, remote access, *etc.*). Such flexibility is not possible in higher or lower levels of abstraction.

The overall IPsec architecture is very similar to previous work [12] and is composed of three modules:

- The data encryption/authentication protocols [1, 2]. These are the “wire protocols,” used for encapsulating IP packets to be protected. Outgoing packets are authenticated, encrypted, and encapsulated just before being sent to the network, and incoming packets are decapsulated, verified, and decrypted immediately upon receipt. These protocols are typically implemented inside the kernel, for performance and security reasons. A brief overview of the OpenBSD kernel IPsec architecture is given in Section 2.
- The key exchange protocol (IKE) [11] is used to dynamically establish and maintain Security Associations (SAs). An SA is the set of parameters necessary for one-way secure communication between two hosts (*e.g.*, cryptographic keys, algorithm choice, ordering of transforms, *etc.*). Although the wire protocols can be used on their own using manual key management, wide deployment and use of IPsec in the Internet requires automated, on-demand SA establishment.

Due to the large number and variety of configurations and options an IKE implementation must support, this part of the IPsec architecture tends to dominate the other two in terms of code size and complexity. The first part of

this paper discusses the OpenBSD implementation of IKE.

- The policy module governs the handling of packets on their way into or out of an IPsec-compliant host. Even though the security protocols protect the data from tampering, they do not address the issue of which host is allowed to exchange what kind of traffic with what other host. While traditional packet filtering mechanisms, such as employed in modern firewalls, can be used (with minor modifications) in enforcing traffic policies, a higher-level mechanism for validating and configuring such filters is needed. The second part of this paper discusses the implementation of a security policy mechanism based on trust management [6] in the OpenBSD IPsec.

1.1 Paper Organization

The remainder of this paper is organized as follows. Section 2 outlines the OpenBSD IPsec architecture. Section 3 gives a brief overview of the IKE protocol, while Section 4 discusses the design and implementation of the OpenBSD IKE implementation, and Section 5 presents the security policy mechanism. Related and future work is presented in Section 6.

2 OpenBSD IPsec

IPsec in the OpenBSD kernel is implemented as just another pair of IP transport protocols (AH and ESP). Thus, incoming IPsec packets destined to the local host are submitted to the appropriate IPsec protocol for processing based on the protocol number in the IP header. The SA needed to process the packet is located in an in-kernel database using information retrieved from the packet itself. Once the packet has been correctly processed (decrypted, authenticity verified, *etc.*), it is re-queued for further processing by the IP module, accompanied by additional information (such as the fact that it was received securely) for use by higher protocols and the socket layer.

Outgoing packets require somewhat different processing. When a packet is handed to the IP module for transmission, a lookup is made in a modified

version of the routing table (called Security Policy Database, or SPD, in the IPsec standards) to determine whether that packet needs to be processed by IPsec. If this is the case, the result of the lookup also specifies what SA(s) to use for IPsec-processing the packet. Once processed, the packet is then re-queued for transmission by IP. If no SA is currently established with the destination host, the packet is dropped and a message is sent to the key management daemon through the **PF_KEY** interface [16]. It is then the key management's task to negotiate the necessary SAs.

To manage the SA and SPD tables, we use the **PF_KEY** interface, which is similar in concept to the routing socket interface available in BSD. Both manual keying utilities and key management daemons (such as IKE or Photuris [13]) use this interface to communicate with the kernel.

A somewhat dated overview of the OpenBSD IPsec architecture is given in [15].

3 The IKE Protocol

IPsec provides a solution to the problem of securing communications. However, for large-scale deployment and use, an automated method for managing SAs and key setup is required. There are several issues in this problem domain: negotiation of SA attributes, authentication, secure key distribution, and key aging to name some. Manual management is complicated, tedious, error-prone, and does not scale. Standardized protocols addressing these issues are needed; IETF's recommended protocol is named IKE, the Internet Key Exchange. IKE is based on a framework protocol called ISAKMP and implements semantics from the Oakley key exchange, therefore IKE is also known as ISAKMP/Oakley.

The IKE protocol is unfortunately a rather complex one, with many modes and options. Furthermore, new extensions proposed result in a further increase in complexity. Interoperation has been a problem because of this, but we are beginning to see good interoperability in the mandatory parts of the protocol.

The IKE protocol has two phases: the first phase establishes a secure channel between the two key

management daemons, while in the second phase IPsec SAs can be directly negotiated. The first phase negotiates at least an authentication method, an encryption algorithm, a hash algorithm, and a Diffie-Hellman [9] group. This set of parameters is called a "Phase 1 SA." Using this information, the peers authenticate each other and compute key material to use for protecting Phase 2. Depending on the protection suite specified during Phase 1, different modes can be used to establish a Phase 1 SA, the two most important ones being "main mode" and "aggressive mode." Main mode provides identity protection, by transmitting the identities of the peers encrypted. Aggressive mode provides somewhat weaker guarantees, but requires fewer messages and allows for "road warrior" ¹ types of configuration using passphrase-based authentication.

The second phase is commonly called "quick mode" and results in a IPsec SA tuple (one incoming and one outgoing). As quick mode is protected by a Phase 1 SA, it does not need to provide its own authentication protection, allowing for a fast negotiation (hence the name). Optionally, a new Diffie-Hellman computation can be done, providing "Perfect Forward Secrecy". PFS is an attribute of encrypted communications allowing for a transient session key to get compromised without affecting the security of future keys negotiated under the same Phase 1 SA (in other words, all session keys are cryptographically independent).

4 OpenBSD IKE

During spring 1998, Ericsson Radio Systems was looking for technology that could secure general IP-traffic in networks of tens, maybe hundreds of thousands of participating hosts. Fairly soon it became evident that IPsec was the right approach, but it was not at all clear what IKE implementation to use. The IKE standard was still evolving, and available implementations were lacking in either functionality, portability, exportability, or scalability. After having been presented with the state of the IKE market, Ericsson agreed to fund the development of an IKE implementation written from scratch, *isakmpd*. The initial authors were Niklas Hallqvist and Niels Provos, both from the OpenBSD project.

¹Remote mobile users that need to access the protected network behind a firewall, using IPsec.

4.1 Architecture

When reading the drafts (later RFCs) on IKE, it became clear the protocol was complex, with many degrees of freedom. It was also known that *isakmpd* would be ported to several platforms, each with different APIs to the IPsec stack. There were also a number of proposals for IKE extensions in varying stages of completion. All these facts pointed towards a very modular architecture with distinct APIs between the subsystems. To avoid development complexity, we also decided to map the concepts of the standards fairly directly onto internal data structures.

Given how *isakmpd* would work (accepting inbound packets, doing some processing in the packet-prescribed context, sending a reply), it felt natural to build a message-based event-driven application. Thus *isakmpd* looks like most Unix UDP servers, with a main loop consisting of a select call followed by a multiplexor calling the right handlers for the occurring events.

The most common event is packet arrival, handled by the message module which is also responsible for packet validation and context lookup. Another fairly common event is the timeout, dealt with by the timer module. There are also application events, which are upcalls from the controlled application, in our case the IPsec stack. The design of *isakmpd* allows for other such "applications" in the future. This is the reason why it is called *isakmpd*, instead of *iked*. IKE is just one possible instantiation of the ISAKMP framework. The upcalls are dealt with by the application module, which to a great extent consists of system-dependent code dealing with the IPsec stack at hand. Currently, there exist three application back-ends, PF_KEY, PF_ENCAP and FreeS/WAN's NetLink API.

For controlling *isakmpd* there are a couple of modules worth mentioning. The "user interface" (UI) module listens for asynchronous events that control different aspects of *isakmpd*, like debugging level, active connections *etc.* This is currently done through a FIFO, but the design allows use of sockets or some other IPC mechanism. There is also a configuration module dealing with configuration file parsing, as well as lookups and overrides (via UI) of configuration entries. Last but not least, there is a policy module controlling what kind of SAs are allowed to be negotiated and by whom (see Section 5).

As ISAKMP is a transport-neutral protocol, there is also a transport module, which is actually an abstract class in an object-oriented view. Since IKE only requires UDP as the transport mechanism, there is just one derived class, the *udp* class. Finally, there is also a low-level network interface module which provides interface-walking, *etc.*

As all ISAKMP packets belong to “exchanges,” we chose to create an exchange abstraction which was mainly a script engine and a data structure accumulating context state to later be carried over into SAs. Therefore, there are exchange and SA modules. They deal with creation, lookup, maintenance, aging, and destruction of these structures. Each exchange has a “script,” which is walked for every packet received or transmitted. This makes it easy to create a source file per exchange type, making the code well modularized.

Independent of what exchange is used, there are a lot of common operations that need to be carried out during a negotiation. For this purpose we created separate modules for authentication, encryption, hash computation, and Diffie-Hellman computation. These in turn need more basic modules, like random number generation, long integer math, group math of both *modP* and elliptic curve kinds, and *X.509* certificate management [7].

Lastly, there are miscellaneous modules dealing with things like dynamic loading of code, logging, *etc.*

4.2 Component Description

- The message module.

This module provides an abstract data-type representing individual ISAKMP messages. Internally, the messages are subdivided and indexed by payload type. Exported functionality consists of creation/destruction, incremental payload addition, parsing, validation, and context lookup of incoming messages, registering of post-send functions, transport-independent send logic, and message debugging. There is also generic SA negotiation logic which is covered in the implementation details section below. The reason for this logic being here is because it is driven by the physical message layout.

- The timer module.

A fairly simple module accepting registration of functions to call at specific times together with their actual parameter. In order to get the functions called, the time module exports a function that calculates the timeout parameter to give the *select* call of the main loop, as well as the actual timer run function. Removal and reporting (for debugging) of timers is also supported.

- The application modules (*app*, *pf-encap*, *pf-key*, *etc.*)

These modules deal with the communication with the application for which *isakmpd* is negotiating SAs. Currently, only one application is supported, IPsec. Communication with it occurs through various system-dependent APIs. Operations that need to be supported include getting a fresh SPI, creating an SA, updating a “larval” SA, grouping SA bundles, and, finally, removing SAs. Also needed is a means for telling the IPsec stack that ISAKMP traffic needs to be unencrypted. In OpenBSD, this is achieved by setting the appropriate *setsock-opt(3)* options in the *isakmpd* socket.

- The network modules (*transport*, *udp* and *if*).

The transport module exports an abstract data-type representing a specific transport. It has an associated function pointer table, just like the common *vtables* that C++ compilers create in order to implement polymorphism. Thus the transport structure is really a base class for the real transport classes. There is just one such class at the moment, the *udp* class. Exported functionality consists of creation/destruction (or rather reference/dereference as they are ref-counted) of transports, getting file descriptors ready for I/O to use in the select loop of *main()*, as well as checking them for I/O possibility afterwards. Message sending and reception methods are exported as well, along with endpoint address determination.

- The UI module.

This module is really just a simple command line interpreter. It conveniently accepts commands asynchronously through a one-way FIFO (named pipe). The commands are rudimentary, one letter with a few parameters each. The existing controls deal with issues like debugging, SA management, and dynamic changes to the configuration database.

```

int (*ike_main_mode_initiator[]) (struct message *) = {
    ike_phase_1_initiator_send_SA,
    ike_phase_1_initiator_rcv_SA,
    ike_phase_1_initiator_send_KE_NONCE,
    ike_phase_1_initiator_rcv_KE_NONCE,
    initiator_send_ID_AUTH,
    ike_phase_1_rcv_ID_AUTH
};

```

Figure 1: The Initiator Main Mode script

- The configuration module.

isakmpd maintains a configuration database consisting of section/tag/value triplets, *i.e.* it maps closely to a well known format called ".INI". This configuration database is primed from the configuration file (.INI-style) at program start, and every time a HUP signal is sent to the *isakmpd* process. It is also possible to dynamically alter the database via the UI module. There is functionality to treat the value of a triplet as a comma-separated list, and easily "walk" that list. Otherwise, ordinary database operations like creation, lookup, and removal of entries are exported.

- The policy module.

See section 5 for a description of this module. This module exports only one function, which is called to validate a combination of SA proposal, remote peer identity, and packet selectors (Phase 2 IDs).

- The exchange module.

A key abstraction in *isakmpd* is the exchange. This is the engine that drives the negotiations towards SA establishment. Exchanges form the context of all negotiations, and closely map to the exchange concept of the RFCs. Every exchange is a well-defined, fixed-length sequence of messages between the two peers. Every individual message also has a well-defined minimum content of payloads. This structure of exchanges lends itself to implementation as a generic finite state machine driven by "scripts" supplied by each exchange type. These scripts provide the actions to execute at message reception as well as before/after message transmission. It is also easy to have a generic "syntax checker" inspecting each message, ensuring the required payloads are present. This module's exported API consists of functions for establishing exchanges when acting as initiator, as

well as setting up exchanges for "incoming" negotiations. There are also several lookup functions, finding exchanges using different criteria.

- The SA module.

Just like the IPsec kernel, *isakmpd* needs to maintain its own SA database. This database actually consists of both ISAKMP SAs, which are the results of Phase 1 negotiations, and application SAs from Phase 2. Every SA has attached DOI-dependent (Domain Of Interpretation) data, should we ever need to support other DOIs than IPsec. The SA structure contains both the on-the-wire representation of the SA, as well as internal per-SA data. SAs are created when the negotiation starts, but are inactive until an exchange finalization routine is run. The SA API is mostly a set of life maintenance functions, *i.e.* creation, ref-counting, expiration setup, and destruction operations. Similar to the exchange module, a fairly versatile set of lookup functions is available.

- The authentication module.

IKE allows for several kinds of authentication. An authentication method needs to provide just three functions: generation of a shared secret the peers derive keys from, encoding of a keyed hash proving the authenticity of the peer, and decoding of such a hash thereby verifying the other peer's authenticity. Currently *isakmpd* supports the mandatory pre-shared key authentication method, as well as certificate based (X.509) RSA signature authentication. We plan to support public key encryption-based authentication in the near future.

- Cryptography and math.

Isakmpd builds upon some basic cryptographic and mathematic components.

– Ciphers.

There is a collection of ciphers which can be used interchangeably to protect the data that goes on the wire. It is natural to implement these ciphers as subclasses to a “crypto” base class, which provides hooks for initialization, cloning, and updating of key state, as well as encryption and decryption of data. The separation of key state management from the actual algorithm applications is important for maintaining cryptographic synchronization between the peers. *isakmpd* implements the following algorithms: DES, 3-DES, CAST, and Blowfish.

- Hashes.

As was the case with ciphers, it is also a design requirement that hash algorithms be easy to alter. Thus, hash algorithms are also implemented as subclasses of a generic hash class, providing a simple API for incremental hash computation of concatenated data.

- Diffie-Hellman.

The Diffie-Hellman algorithm is a means of establishing a shared secret between two peers without exposing sufficient data for wire-tappers to compute that secret. The API is simple, since only two functions are needed: creation of a local random big-integer, and computation of the actual secret based on the local big-integer and a similar-type value received from the peer.

- Group mathematics.

The mathematical basis for Diffie-Hellman is called group math. Groups are big-integer arithmetic systems with a few parameters. It turns out that groups are also suitable to implement in an object-oriented fashion, as there are different algorithms that comply with the group math requirements. In *isakmpd*, there is support for two kind of groups, elliptic curves and *modP* groups.

- Big integer mathematics.

Both group mathematics and the public key cryptography used in the authentication and policy modules, need big-integer math. We currently use OpenSSL’s BN functions as well as a few supplementary routines written by us. We have however made the underlying math library exchangeable so other math libraries can

be used if needed. We currently support FSF’s GMP but we also intend to take advantage of hardware support for big-integer operations, since such products have begun to make their appearance in the market.

- The dynamic loader module.

Perhaps a less obvious component to have in a daemon like *isakmpd* is a module for dynamic loading and linking of code. The reason for this module is mainly due to the RSA patent; we cannot ship RSA code in OpenBSD as the license-free implementation cannot be imported to the United States. Therefore, we dynamically load that support if it is available (the supporting libraries can be fetched separately, different versions for different countries). This module exports a function that takes a dynamic load script, written in a very simple language we designed, that describes what files should be loaded and what symbols should be resolved.

- The log module.

Logging is crucial in security applications. It is also important that developers of security software are presented with debugging tools that help them find bugs faster. We consider logging to be such a tool, if it can be controlled in a fine-grained way. This module exports functions to change the levels per logging class, to control where logging information goes and, naturally to actually log both binary and textual buffers.

- The system-dependent module.

In order to maintain portability, every function that may need differing implementations depending on the platform, needs to be placed in a central, exchangeable, system-dependent module. Most often, functions placed here are glue or proxies.

4.3 Implementation Details

4.3.1 The Exchange Script Machine

An IKE exchange normally consists of a fixed number of well-defined messages, which each peer sends every other turn. Recognizing this simple fact, we chose to build the state machine around an engine which ran “scripts” unique for each exchange type. An example of a script is shown in figure 1.

This is the script an initiator runs when doing a “main mode”. The elements of the script are functions, alternately constructing a message to be sent, or dealing with a message that has been received. Along with this semantics description there is also a syntactic “script”, which may look like figure 2. This syntax description describes what payloads are mandatory in each message of the exchange. It also marks when the exchange ends.

4.3.2 Configuration

Configuring IKE is an involved process, due to IKE being a complex protocol. When we were faced with the problem of how to design the configuration language we tried a few simplistic approaches, but they soon turned out to be too inflexible. Thus we decided to use a rather generic configuration syntax which we could fit in everything we wanted. The syntax would also allow for easy dynamic modification of the internal configuration information without reloading a full file. The caveat is that our configuration syntax maps much better to the machine and protocols than to a human being administering *isakmpd*. Our plan was to get someone else write a “real” configuration file format that could be translated into our style. So far no one has taken the bait. Note that ideally, very little configuration should be needed for *isakmpd*; most of the information should be provided on-the-fly by the kernel (at least in the end-to-end case), or through some security policy discovery mechanism.

The file format is commonly known as .INI-format, and a snippet is shown in figure 3. Internally, everything is treated as (section, tag, value) triplets, where the values can optionally be lists of scalar values. The values themselves are often section names thereby giving a tree (or rather a forest) structure to the data.

As we have already mentioned, the internal configuration is dynamically alterable. We saw a need for several “users” altering the configuration concurrently, so we made the API transactional. Each transaction can contain several modifications to the configuration, and they are atomically introduced.

Internally there is also an API to get the actual configuration values. Because of this, it is considered very easy to move the configuration database into other internal formats or even externalize it.

4.3.3 Portability Considerations

From its conception, there was a portability requirement in *isakmpd*. It should run on various platforms, and with different IPsec stacks. Because of this demand, the “sysdep” module was introduced. Each platform we support needs to provide its own version of this module. In principle, all of the IPsec API could be dealt with here, but as APIs can be shared among several platforms (and there even exist standards now), most often the sysdep module only has stub code to call the right API module, like PF_KEY.

PF_KEY may become a standard, but it is only an API for maintaining SAs, and IPsec also needs policy maintenance. All PF_KEY systems we support have chosen to add policy extensions to PF_KEY because of the fact that the API is flexible enough to pass such data as well, and it is easier to extend something working than to invent something entirely new. However, extensions tend to be platform specific, so the PF_KEY support code in *isakmpd* has to deal with several different variants of the protocol. This problem is recognized, and there actually is some consensus between OpenBSD, KAME, and FreeS/WAN that this needs to change, and that the extensions need to converge, if not even be standardized.

With respect to differences in the build environment, we have seen a need to support both main “make” dialects, BSD and GNU. This is of course less than optimal, but given the alternatives it is currently our best option. Furthermore, every supported platform has to provide a makefile fragment wherein constraints on what *isakmpd* should support on that particular platform can be expressed, as well as instructions on how to build system-dependent code.

4.3.4 Debugging Support

Being a security critical application, it is vital *isakmpd* be as bug-free as possible. All software contains bugs, and all development creates new ones. Recognizing that, we have chosen to make debugging a more pleasant task than it usually is. Normally *isakmpd* detaches from the controlling terminal and logs only exceptional conditions to the syslog facility. However, in order to be able to run under a normal debugger, it is possible to run in

```

int16_t script_identity_protection[] = {
    ISAKMP_PAYLOAD_SA, /* Initiator -> responder. */
    EXCHANGE_SCRIPT_SWITCH,
    ISAKMP_PAYLOAD_SA, /* Responder -> initiator. */
    EXCHANGE_SCRIPT_SWITCH,
    ISAKMP_PAYLOAD_KEY_EXCH, /* Initiator -> responder. */
    ISAKMP_PAYLOAD_NONCE,
    EXCHANGE_SCRIPT_SWITCH,
    ISAKMP_PAYLOAD_KEY_EXCH, /* Responder -> initiator. */
    ISAKMP_PAYLOAD_NONCE,
    EXCHANGE_SCRIPT_SWITCH,
    ISAKMP_PAYLOAD_ID, /* Initiator -> responder. */
    EXCHANGE_SCRIPT_AUTH,
    EXCHANGE_SCRIPT_SWITCH,
    ISAKMP_PAYLOAD_ID, /* Responder -> initiator. */
    EXCHANGE_SCRIPT_AUTH,
    EXCHANGE_SCRIPT_END
};

```

Figure 2: The syntax of an ID_PROT exchange

Incoming phase 1 negotiations are multiplexed on the source IP address.

```

[Phase 1]
192.168.0.1= ISAKMP-peer-node-0

[ISAKMP-peer-node-0]
Phase= 1
Transport= udp
Address= 192.168.0.1
Configuration= Default-main-mode
Authentication= yoursharedsecretwith0

[Default-main-mode]
DOI= IPSEC
EXCHANGE_TYPE= ID_PROT
Transforms= 3DES-SHA,3DES-MD5

[3DES-SHA]
ENCRYPTION_ALGORITHM= 3DES_CBC
HASH_ALGORITHM= SHA
AUTHENTICATION_METHOD= PRE_SHARED
GROUP_DESCRIPTION= MODP_1024
Life= LIFE_600_SECS

[LIFE_600_SECS]
LIFE_TYPE= SECONDS
LIFE_DURATION= 600,450:720

```

Figure 3: Configuration entry samples

the foreground, sending logging messages to *stderr* instead. As we have already mentioned, the logging module has a fine-grained control mechanism making it easy to choose detailed information on certain topics. In order to ease problem pinpointing, almost every intermediary computation can be logged.

The build environment also contains instructions on how to build *isakmpd* with two different memory allocation debugging tools: ElectricFence, for finding buffer overflows and use after deallocation, and Boehm's garbage collector to find memory leaks. We periodically run with these tools to test for such problems.

4.3.5 Addressing Denial of Service Attacks

IKE is subject to DoS (Denial of Service) attacks since state has to be kept in the responder after the first message has been received. If a malicious peer starts flooding *isakmpd* with exchange initiations, a lot of state will accumulate in the responder. Worse yet, in aggressive mode, the responder will have to do expensive computational work² before the peer has been authenticated. These issues are actually protocol problems and could have been moot, if only the "cookie" mechanism adopted from the Photuris protocol had been understood and used correctly [13, 17]. Since the protocol has been standardized, we need to address the potential attacks. Our approach is twofold: first off, we always check memory allocation for failure, and back out, cleaning up all resources tied in with the message we are re dealing with. Second, we use a maximum, configurable, exchange lifetime. If the exchange times out, all resources are given back to the system.

We have considered additional measures, like aggressive random tail drop of exchanges stuck in the state after the first reply. This would be somewhat analogous to the normal response to TCP SYN-floods.

4.3.6 Solving the RSA "problem"

At the time we started implementing *isakmpd*, exporting a US RSA implementation in source form to the world at large was illegal. Another problem was

²Even hardware accelerators for big number computation cannot handle the high volume of operations that would be involved in such a DOS attack.

that it is not legal to use the RSA algorithm within the US unless one has a license from RSA Inc. or use the US-originated non-commercial RSAREF library. Thus, there was no way to make a distribution that would be free to use both in the US and in the rest of the world, because the only implementation that is free in the US was not exportable. OpenBSD has solved this problem in other places of the source tree in an elegant way: we chose to use all RSA functionality via a dynamically linked shared library, *libcrypto*, which is part of OpenSSL. This library exists in three variants: one RSA-crippled, with no RSA support at all, one with internationally written RSA code and one with RSAREF. We ship the RSA-crippled version as that one has no patent or exportability issues at all. Then we tell international users to fetch the international *libcrypto* version, and US users to get the one based on RSAREF (if they meet criteria to legally use it).

This could work for *isakmpd* too, if it were not for the fact that we want *isakmpd* to be statically linked, so we can get IKE negotiation capabilities really early in the boot process.

The solution was to use dynamic linking via the *dlopen* API. Every RSA-related symbol of *libcrypto* needs to be accessed indirectly through a pointer. This pointer is initialized with the address of the statically linked RSA-crippled stubs. After a successful dynamic link the pointers get reset to the newly loaded *libcrypto* equivalents. It is not considered a fatal error if the dynamic linking fails. Not all operating systems allow statically linked binaries to use *dlopen* though, but those who do can benefit from this.

4.3.7 Performance and Code Size

The SA negotiation is very CPU-intensive. More specifically, in main and aggressive mode there is always a Diffie-Hellman exponentiation and sometimes, depending on authentication method, RSA or DSS signature operations that are fairly expensive in terms of CPU processing. In quick mode, the DH exponentiation is optional but recommended. That exponentiation is what provides "Perfect Forward Secrecy." Some sample timings can be found in figure 4.

In its current state, *isakmpd* consists of roughly 36,000 lines of code, almost all of it in C. This in-

Exchange	Seconds
Main mode, 3DES, SHA, DH group 2, pre-shared key	1.44
Quick mode, 3DES, SHA, PFS (DH group 2)	1.40
Main mode, DES, MD5, DH group 1, pre-shared key	0.95
Quick mode, DES, MD5, PFS (DH group 1)	0.60
Aggressive mode, 3DES, SHA, DH group 2, RSA signature (X.509)	1.50
Quick mode, 3DES, SHA, no PFS	0.35

Figure 4: A Pentium 200MHz running two instances of isakmpd negotiating over the loopback interface (an exchange between two distinct machines may actually finish faster as some computations can be carried out in parallel).

cludes commentary, which we have at least tried to be fairly generous with. Security protocol implementations need to be auditable, and readability is therefore an important aspect. 4,000 of these are the platform-dependent parts, and 2,500 are regression testing. The static memory footprint for i386 is approximately 950KB for a full-blown version and 300KB for a trimmed down version with support only for mandatory ciphers, exchanges, groups, and authentication methods (no debugging or refined policy handling is included in the trimmed-down version).

5 Security Policy

When discussing security policy, it is often useful to define the term in the appropriate context. For our purposes, security policy in the network layer is the information needed to decide whether a packet should be accepted/forwarded or dropped. Further restricting the definition in the IPsec context, security policy dictates what classes of packets are acceptable over a specific SA. This is all the more important for IPsec, since the encapsulation mechanism used literally allows establishment of arbitrary virtual topologies over the network fabric.

Since there exists no standard mechanism for specifying, disseminating, and processing security policy for IPsec, we have adopted some ongoing research work based on a compliance-checking architecture. The concept behind this architecture is that, at SA establishment time, we utilize some mechanism that validates the suitability of an SA for a particular class of packets and a remote principal at IKE exchange time; all the characteristics of the SA (cryptographic algorithms, key sizes, transform ordering, etc.), along with the packet classes (in effect, a set of

packet filter rules) and the remote principal's identity (public key, X.509 certificates, passphrase, etc.) are available at that stage. It is important to realize that this operation is performed only infrequently compared to the number of packets that will use the established SAs. Thus, it is possible to use a mechanism that is more general, powerful, and extensible than a simple packet filter specification language. We would also like to be able to utilize credentials delegating authority, as we have found these to allow easier and more scalable administration.

The higher-level mechanism for security policy compliance-checking we use is a trust-management system. Trust-management systems [5, 4] provide a unified approach to specifying security policies, credentials, and relationships between principals in the system. Unlike traditional certification schemes, trust-management credentials bind keys directly to the authorization to perform some task. A trust-management system provides a highly-adaptable general-purpose mechanism for specifying security policies and credentials. A principle of trust management is "monotonicity." This means that policies and credentials can only have a positive effect on the privileges of a principal; it is not possible to revoke privilege by issuing a credential. This may only be done by expiring credentials, or by modifying the relevant policies and credentials. For an extensive overview of trust-management, see [3].

KeyNote is an instantiation of a trust-management system, designed to be simple yet flexible. It provides a single language for both policies and credentials, based on predicates that describe the trusted actions permitted by holders of specific public keys (or other cryptographic identifiers). For more details on KeyNote syntax and processing, see [4]. For more details on the policy architecture itself, see [6]. The following subsection discusses some implemen-

tation specifics.

5.1 Implementation Details

Modifying *isakmpd* to make use of the compliance-checking architecture for policy resolution proved straightforward. *isakmpd* was initially designed with a rudimentary mechanism for verifying security associations proposed by the remote peer. The set of acceptable security associations was read from the configuration file, and then consulted when examining the proposed SA. However, this scheme lacked flexibility and extensibility. In particular, it was not possible to delegate authority, allow for very fine-grained SA specification without an explosion in the size of the configuration file, take into consideration information not directly relevant to the SA (such as time of day, or system security level), nor allow for flexible packet selectors (an exact match was required).

Since this verification mechanism was implemented as a procedure call, we only had to modify the invoking code to call another procedure that ultimately invoked KeyNote. This change occurred in two places:

1. When the Responder of an IKE exchange examines the list of IPsec (Phase 2) SAs to determine which one is acceptable.
2. When the Initiator receives (during Phase 2) the response containing the acceptable SA.

When invoked, the procedure converts information taken from the *exchange* and *sa* structures to a format suitable for use by KeyNote. Such information contains the IPsec protocols to be used, the cryptographic algorithms to be used, the packet selectors requested (Phase 2 User IDs), the cryptographic identifier used in Phase 1 by the remote peer, *etc.*

This cryptographic identifier is used by the compliance checker to determine which part of the security policy is relevant to a specific request. If public key authentication was used, then our security policy may directly refer to said public key, and the same applies for passphrase authentication. For X.509-based authentication, we have a number of options as to who policy may refer to:

- The public key of the remote principal as it appears in the Subject field of the X.509 certificate, or the X.509 certificate itself. This form of delegation is the most direct and limited in scope.
- The public key or X.509 certificate of some certification authority (CA) that ultimately “speaks for” the remote principal. This may be the CA immediately validating said principal, or some other CA further up in a CA hierarchy. The higher up the CA we delegate to, the broader the scope of the delegation (and thus, more users share the same rights). Note that it is possible to delegate a set of rights to some CA that “speaks for” some user, and simultaneously give more rights to that specific user. Reducing a user’s privileges through the same mechanism is not feasible under KeyNote, however (because of monotonicity, as previously described).
- Since public keys and X.509 certificates can be cumbersome to manipulate even in a text form, it is possible to use the Distinguished Name as it appears in an X.509 certificate. This makes policies much more concise and readable. An added benefit is that certificates (and even keys) may change without affecting the policy (although in some cases this may turn into a liability). We can use the DN of the remote principal directly, or that of some CA that “speaks for” the principal.

The assembled information is passed on to KeyNote, and the response indicates whether the SA should be accepted or dropped. In effect, KeyNote is verifying that the combination of remote peer, IPsec protocols (and algorithms, lifetimes, *etc.* used by those protocols), and packet selectors are acceptable by policy. This policy may be expressed solely in terms of local policy or as a combination of local policy and (signed) credentials. These credentials may be acquired during the Phase 1 exchange (provided by the remote peer) or at any point in time afterwards (*e.g.*, fetched on-demand through some out-of-band protocol³). As soon as an SA is accepted, the search is concluded.

The procedure is called once for each distinct SA proposal received from the peer (since there is no

³We have experimented with fetching credentials from a web server, using a primitive cgi-script and a database keyed on public keys and X.509 Distinguished Names.

way to efficiently encode all the SA proposals in one action attribute set and have KeyNote make a decision on which one to select – this is a drawback of using KeyNote instead of a more complex policy language). Note however that each such invocation is very “lightweight” in processing terms: converting the relevant information is straightforward, and any cryptographic operations are only performed once and their results cached for future use. The policy assertions are loaded once at startup time (and reloaded if *isakmpd* is asked to re-initialize). Some simple experiments show that the cost of invoking KeyNote increases linearly with the number of assertions in use, and that for a simple setup of 3-4 assertions/credentials the cost is in the order of 150µsec.

Here, we wish to make two additional observations:

- KeyNote is invoked during Phase 2 only. While it is trivial to allow policy control over establishment of Phase 1 SAs, we believe that this is both unnecessary and potentially confusing to users. Since Phase 1 SAs are used only by *isakmpd* and have no direct effect on the system or on network traffic, this approach does not compromise safety.
- Currently, compliance checking on the initiator is performed when the accepted SA is received from the responder (message 2 in Quick Mode). Ideally, this check should be done before transmission of the first message in Quick Mode, to avoid transmitting SA proposals that in the end will not be accepted by us. Processing after receipt of message 2 should be limited to verifying that the returned SA is among those offered in the first message. We elected not to do this because of code complexity: because KeyNote support was added after most of *isakmpd* was written, the code that constructs the list of SAs in message 1 was already intricately tied to message construction, configuration file parsing, and attribute syntax verification. Rewriting the relevant code just to accommodate KeyNote would involve serious restructuring. We intend to rewrite that piece of *isakmpd* in the near future to retrieve SA information from the kernel (as opposed to a configuration file). At that time, an interface better suited to policy compliance checking will be introduced. We should note that this issue is not an artifact of our use of KeyNote; using any security policy system on the initiator side

would require the same code restructuring.

In terms of code size, the “glue” code between *isakmpd* and KeyNote was about 1200 lines, almost exclusively dealing with the conversion of information from *isakmpd*’s internal structures to KeyNote action attributes. We also had to add about 50 lines of code in different parts of KeyNote, dealing with initialization and record keeping. The code displaced by KeyNote was approximately 500 lines long. The KeyNote library itself is about 5000 lines (not including the cryptographic functions, where *libcrypto* is used).

6 Conclusion

6.1 Current State

We believe that *isakmpd* currently addresses all mandatory features in the RFCs. We also implement most optional features. *isakmpd* currently runs on OpenBSD’s old IPsec stack with PF_ENCAP, OpenBSD’s current stack with PF_KEY, FreeBSD/WAN with Linux NetLink API and FreeBSD/NetBSD with KAME’s IPsec stack via PF_KEY. We have also made it possible to shave off much of the extras at compile time, thus making *isakmpd* a candidate for being used in small embedded systems. *isakmpd* is in production used in numerous sites.

6.2 Future Directions

There seems to be an increasing number of proposed new IKE extensions after every IETF. We are, however, reluctant to incorporate them all as code bloat is a problem we should fight to maintain any kind of security. Something we definitely are going to add is IPv6 support, as we recently have started shipping OpenBSD with an IPsec-aware IPv6 stack. Other likely enhancements are support for PKCS#11 (an API to talk to cryptographic tokens, like smartcards, for authentication), challenge-response authentication for Phase 1 exchanges and PKIX compliance. A major short-term project is support for cryptographic hardware for RSA and Diffie-Hellman computation, since OpenBSD has begun to sup-

port a cryptographic services framework in the kernel. Other minor projects involve integration with DNSSEC [10] infrastructure once we see further deployment and use, and “New group mode” support to dynamically negotiate new groups to compute DH secrets in. There are plans to support some new platforms, for example FreeS/WAN over PF_KEY and Solaris 8. There are other commercial Unices with IPsec stacks which we may port *isakmpd* to. Closer integration with the kernel and userland applications (possibly through the *setsockopt(3)/getsockopt(3)* API), and various projects involving policy discovery/negotiation (in particular, direct exchanging of KeyNote credentials) and automatic configuration are also part of our plans for future work.

6.3 Interoperability

We have attended a couple of interoperability workshops as well as carried out our own tests and have succeeded remarkably well, given the complexity of the IKE specifications. A lot may be attributed to our flexible configuration which, however, cannot be said to be user-friendly. We have been known to interoperate with the 3com Pathbuilder 500, Ashley Laurent VPCOM, Axent Raptor, Cendio Fuego Firewall, CheckPoint FireWall-1, Cisco IOS, Cisco PIX, F-secure VPN+, FreeBSD/NetBSD KAME, Intel LanRover, Linux FreeS/WAN, Nortel Contivity, PGP VPN, Radguard cIPRO, Teamware TWISS, Windows 2K, and Timestep Permit.

Most of this interoperation has been with pre-shared keys. Unfortunately we have not yet had a chance to do extensive certificate-based interoperability testing.

6.4 Security Considerations

As might have become clear by now, IKE is a complex protocol, perhaps overly so. As we are implementing security, complexity is not something well looked upon. Complex protocols are implemented with complex programs which tend to have more bugs, and some bugs might just happen to be security breaches. Modular design with clear APIs internally helps reduce complexity and allows for easier auditing, but there is still a lot more risk with complex programs than with simple ones. There are

simpler alternatives to IKE, more limited in functionality, but likely more secure [13].

6.5 Related Work

There are of course other Open Source projects that implement IKE, the two most widely known being the Linux FreeS/WAN project's *Pluto*, and *Racoon*, of the KAME project whose IPsec stacks exist for both NetBSD and FreeBSD. Both of these are only meant for their respective platforms, unlike *isakmpd*, which is meant to be a portable implementation. As a matter of fact, *isakmpd* runs on top of both the FreeS/WAN and KAME stacks. *Racoon* is, to our knowledge, the only IKE implementation with IPv6 support. There are also other key-management protocol implementations available, an example is *photurisd*, OpenBSD's Photuris implementation. An extensive overview of the employment of cryptography in OpenBSD may be found in [8].

7 Acknowledgments

We would like to thank Matt Blaze, Theo de Raadt, Martin Fredriksson, Markus Friedl, Hugh Graham, John Ioannidis, Håkan Olsson, Niels Provos, and Jonathan Smith for their support, comments, suggestions, and work in various aspects of this project and paper. Most of the development of *isakmpd* was funded by Ericsson Radio Systems. The security policy work mentioned in this paper was supported by DARPA under grant F39502-99-1-0512-MOD P0001.

8 Availability

All the software described in the paper is available through the OpenBSD web page at:

<http://www.openbsd.org/>

OpenBSD is based in Calgary, Canada. All individuals doing cryptography-related work do so outside countries that have limiting laws.

References

- [1] R. Atkinson. IP Authentication Header. RFC 1826, August 1995.
- [2] R. Atkinson. IP Encapsulating Security Payload. RFC 1827, August 1995.
- [3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag Inc., New York, NY, USA, 1999.
- [4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The keynote trust management system version 2. Internet RFC 2704, September 1999.
- [5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proc. of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, Los Alamitos, 1996.
- [6] M. Blaze, J. Ioannidis, and A. Keromytis. Trust Management and Network Layer Security Protocols. In *Proceedings of the 1999 Cambridge Security Protocols International Workshop*. Springer, 1999.
- [7] Consultation Committee. *X.509: The Directory Authentication Framework*. International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.
- [8] T. de Raadt, N. Hallqvist, A. Grabowski, A. D. Keromytis, and N. Provos. Cryptography in OpenBSD: An Overview. In *Proc. of the 1999 USENIX Annual Technical Conference, Freenix Track*, pages 93 – 101, June 1999.
- [9] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov 1976.
- [10] D. Eastlake and C. Kaufman. Dynamic Name Service and Security. Internet RFC 2065, January 1997.
- [11] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments (Proposed Standard) 2409, Internet Engineering Task Force, November 1998.
- [12] John Ioannidis and Matt Blaze. The Architecture and Implementation of Network-Layer Security Under Unix. In *Fourth Usenix Security Symposium Proceedings*. USENIX, October 1993.
- [13] P. Karn and W. Simpson. Photuris: Session-key management protocol. Request for Comments (Experimental) 2522, Internet Engineering Task Force, March 1999.
- [14] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, November 1998.
- [15] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom) '97*, pages 1948 – 1952, November 1997.
- [16] D. McDonald, C. Metz, and B. Phan. PF_KEY Key Management API, Version 2. Request for Comments (Informational) 2367, Internet Engineering Task Force, July 1998.
- [17] W. A. Simpson. IKE/ISAKMP Considered Harmful. *USENIX ;login.*, December 1999.

Transparent Network Security Policy Enforcement

Angelos D. Keromytis
Distributed Systems Lab, University of Pennsylvania
angelos@openbsd.org

Jason L. Wright
Network Security Technologies, Inc. (NETSEC)
jason@openbsd.org

Abstract

Recent work in the area of network security, such as IPsec, provides mechanisms for securing the traffic between any two interconnected hosts. However, it is not always possible, economical, or even practical from an administration and operational point of view to upgrade the software and configuration of all the nodes in a network to support such security protocols.

One apparent solution to this problem is the use of security gateways that apply the relevant security protocols on behalf of the protected nodes, under the assumption that the “last hop” between the security gateway and the end node is safe without cryptography. Such a gateway can be set to enforce specific security policies for different types of traffic. While this solution is appealing in static scenarios (such as building so-called “intranets”), the use of Layer-3 (network) routers as security gateways presents some transparency and configuration problems with regards to peer authentication in the automated key management protocol.

This paper describes the architecture and implementation of a Layer-2 (link layer) bridge with extensions for offering Layer-3 security services. We extend the OpenBSD ethernet bridge to perform simple IP packet filtering and IPsec processing for incoming and outgoing packets on behalf of a protected node, completely transparently to both the protected and the remote communication endpoint. The same mechanism may be used to construct “virtual local area networks,” by establishing IPsec tunnels between OpenBSD bridges connected geographically separated LANs. As our system operates in the link layer, there is no need for software

or configuration changes in the protected nodes.

1 Introduction

Network bridges are simple devices that transparently connect two or more LAN segments by storing a frame received from one segment and forwarding it to the other segments. More intelligent bridges make use of a spanning tree algorithm to detect and avoid loops in the topology. We have implemented the basic form of an ethernet bridge in OpenBSD that also provides an IP filtering capability. Thus, the bridge can be used to provide a LAN-transparent firewall between hosts such that no configuration changes are needed on client machines, and only minor changes in network topology are necessary.

For this, we make use of *ipf*, the standard packet filtering mechanism available. As ethernet frames pass through the bridge, they are examined to see if they carry IP traffic. If not, the frame is just bridged. If the frame does contain IP traffic, the ethernet header is removed from the frame and copied. The resulting IP packet is passed on to *ipf*, which notifies the bridge whether the packet is to be forwarded or dropped. The ethernet header of the frame under examination is appropriately modified on the frame to be forwarded, and the resulting frame is then bridged as normal.

The bridge can also be used to enforce restrictions on which addresses can appear on each ethernet segment, which helps localize where ARP spoofing attacks can occur. Static MAC address cache entries are provided so hosts can be limited to a particu-

lar port and malicious users cannot force the bridge to send traffic to the wrong segment. The ability to learn MAC addresses dynamically is configurable on each port of the bridge, and broadcast discovery for machines unknown to the bridge can be toggled on a per port basis. Additionally, a mechanism is provided for filtering ethernet frames based on source and/or destination MAC address.

This functionality, useful on its own, can be coupled with the IPsec [9] support available in OpenBSD, to allow creation of virtual LANs. This is achieved by overlaying an IPsec-protected virtual network on the wide area network (or even the Internet itself). The changes necessary to the bridge and IPsec code for this were fairly minimal, due to compatibility of some design decisions made independently in the development of the two packages.

The enhanced bridge can also be used to provide transparent IPsec gateway capability for a host or even a network. In this mode, the bridge examines transient IP traffic and may, depending on security policy, establish IPsec security associations (SAs) with a remote host pretending to be the local communication endpoint for an IP session¹. There are two main benefits from this. First, this allows protection of the communications of a host or network without changes to the protected hosts (which may not even be possible, for old, unsupported, or extremely lightweight systems). Second, the security gateway can act as a security policy enforcer, ensuring that incoming and outgoing packets are adequately protected, based on system or network policy.

1.1 Paper Organization

Section 2 briefly describes the bridge itself and the filtering of frames containing IP traffic. Section 3 describes the use of IPsec in conjunction with the bridge to build virtual LANs and transparent IPsec gateways. Section 4 discusses open ends and future work, and Section 5 concludes the paper.

¹The term "IP session" is used here loosely to imply a packet flow between two hosts, one of which is on one of the local segments and is "protected" or "supervised".

2 Bridge

Bridges are devices that operate at the data link layer, tying together different ethernet (or other LAN) segments. In OpenBSD, the bridge is implemented as a pseudo-network interface inside the kernel. Real ethernet interfaces are added to the bridge interface as "bridge members," and for the purpose of using IPsec with the bridge, *enc* interfaces can be added on as members. The *enc* interfaces contain the security association (SA) for communication to remote LANs. In all ethernet drivers under BSD, received frames are assembled into mbufs [11], a data structure that provides for easy insertion and deletion of data with little or no data copying. The ethernet header is removed and passed along with a reference to the receiving interface and the mbuf containing the frame data to *ether_input()*. The bridge intercepts the frame early in this function, after a small amount of bookkeeping is performed.

On entry to the bridge code, the frame is checked to see if it is a broadcast, multicast, or unicast frame (Figure 1). Broadcast and multicast frames are copied and queued on the bridge (so they can be forwarded in all member interfaces), and the original frame is returned to *ether_input()*, so that it can be processed by the bridge machine itself. Unicast frames are checked to see if the destination matches any of the MAC addresses of ports on the bridge; if so, the frame is returned to *ether_input()* for local processing. If the frame is unicast and addressed to the bridge machine, the frame is queued and not passed back to *ether_input()*. When a packet is queued, a software interrupt is scheduled so that bridge processing will occur outside of the interrupt context of the ethernet card.

The bulk of the frame processing occurs in the software interrupt handler, *bridgeintr()* (see Figure 2). This routine loops through each bridge interface, pulling frames from their input queues. The source ethernet address and source interface are recorded into the bridge's address cache for each frame (after some address spoof-checking). The destination ethernet address is looked up in the cache; if the interface returned by the lookup is the same as the interface where the frame originated, no further processing is done. If the destination interface differs from the source, the frame must be forwarded (bridged). If the frame is for a multicast or broadcast destination, the frame must be forwarded to all member interfaces of the bridge. To avoid overloading *enc*

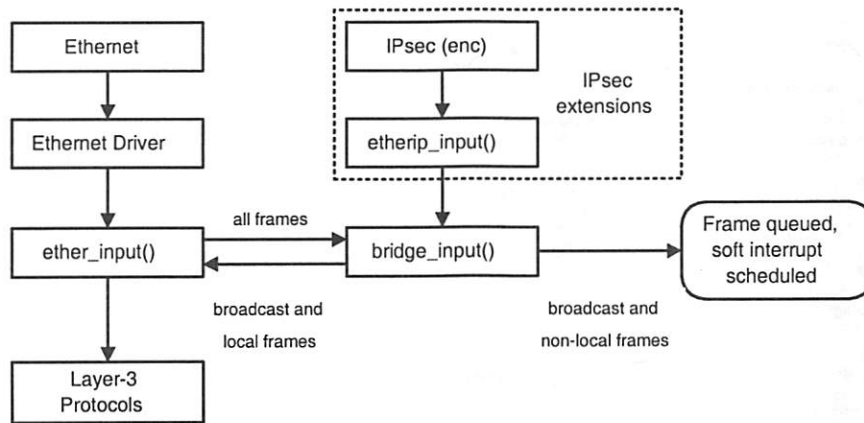


Figure 1: Frame flow from driver to bridge and layer-3 protocols.

interfaces with multicast traffic from fast ethernet interfaces, it is possible to disallow multicast packet and/or frame forwarding over the bridge. Currently, this is specified for the whole bridge. In the future, we would like to be able to specify this on a per-member interface basis.

2.1 Layer-3 Filtering

Before frames are forwarded, they are filtered by calling one of the *ipf* routines with the frame to be processed. This allows for standard filtering rules to be applied to bridge member interfaces as they would be for normal routed firewall. Rules are applied to all incoming frames that contain IP traffic and are bound to each member interface.

The *ipf* routines expect an IP packet to be passed to them, but the bridge operates in terms of ethernet frames. The ethernet header is examined to determine whether the frame contains an IP packet. Since there are two possible encapsulation methods for IP over ethernet, both must be examined and the appropriate amount of header information must be copied and removed from the frame, leaving the IP data intact. The resulting packet is passed to *ipf*, which either drops the packet or returns it. Packets that are not filtered have their ethernet headers reattached and are finally forwarded as determined by the bridge. Using this approach, we avoided having to modify *ipf* code at all.

2.2 Layer-2 Filtering

In addition to providing IP (Layer-3) filtering, the bridge is capable of filtering packets based on source and destination ethernet MAC address. The filtering rules follow a syntax much like the *ipf* rules and are applied in the order in which they are added. Rules can be applied both as a frame is received by the bridge (on input) or before the frame is sent out from the bridge (output).

The bridge can also be used to block all non-IP traffic. A flag on each member interface specifies whether it should allow non-IP traffic to be passed in or out based on the protocol field in the ethernet header. This allows frames to be blocked when they cannot be filtered by the Layer-3 mechanisms provided so that tunnels through other protocols cannot be created. The only protocols allowed through an interface with this flag are the protocols necessary for IP to function: IPv4, IPv6, ARP, and RARP.

2.3 Bridge as Normal Host

A machine acting as a bridge need not have an IP address. All of the filtering provided by the bridge and *ipf* can be handled in the absence of an IP address, and this is actually an easier case to handle.

For the bridge machine to act as a normal host, in addition to its duties as a bridge, several changes were necessary to the path a frame takes through the kernel. As discussed above, unicast frames that

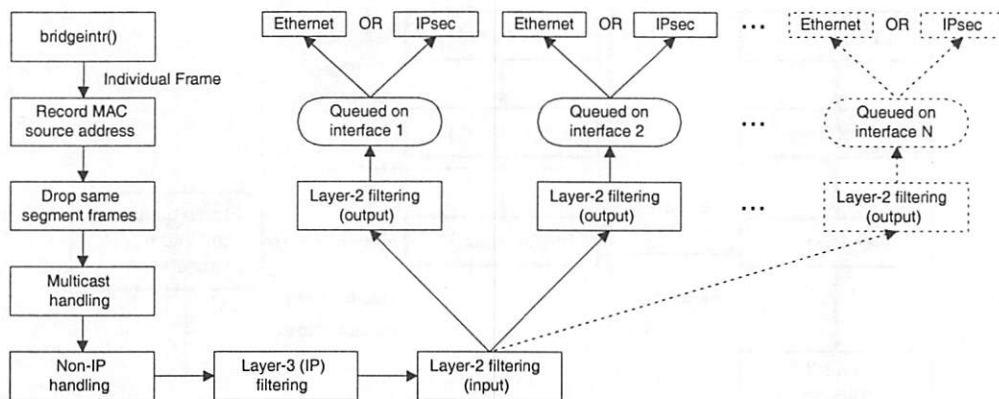


Figure 2: Frame flow from within *bridgeintr()* with Layer-2 and Layer-3 handling.

are addressed to any of the member interfaces of the bridge are simply returned to *ether_input()*. Broadcast and multicast frames must be copied. The original frame is returned to *ether_input()*, and the copy is queued on the bridge.

For frames sent by the bridge, *ether_output()* was modified to include a special case for interfaces that are bridge members and the frame to be sent is passed to *bridge_output()*. This function examines the ethernet destination address of the frame. For unicast destinations, the bridge address cache is used to locate the recipient. For multicast and broadcast destinations, as well as unicast destinations not found in the address cache, the frame is forwarded to all member interfaces of the bridge.

As a result of this design, a machine acting as a bridge can also participate on the LAN as a normal host. When, for example, it sends an ARP request for a host, it will be forwarded out of every member interface. When a reply is received on any interface, the source interface and address are added to the bridge address cache as well as its ARP cache, and the frame is processed as normal. From there, all unicast frames to the remote host will use the information from the address cache for sending frames only on the correct interface.

2.4 Bridge Security

As discussed previously, the bridge provides several methods for enforcing network security policy. One form of internal attack is MAC spoofing where one host forges packets using the ethernet MAC address

of a victim host. The bridge provides two measures for preventing this attack from being completely successful: Layer-2 filters and static address entries.

For the Layer-2 filters, the ethernet MAC address of the potential victim is added to a set of rules. For the bridge interface on the segment where the host is supposed to be, rules are added to permit the address to be the source and destination of frames for input and output. On the other interfaces, the address is added to rules blocking it as a source address on input and destination address on output from each interface.

Additionally, adding a static address cache entry that binds the ethernet MAC address of the potential victim host to the bridge interface on the same segment as the host will prevent the bridge address cache from being polluted with invalid data. The bridge cannot prevent the attack from being successful on individual segments, but it can limit its scope in one segment only.

Another form of internal attack, ARP spoofing, involves a host on the network using its own MAC address and forging ARP responses claiming to be another host. The bridge does not treat ARP packets different from other packets, so this attack is not directly preventable. The attacking host may be able to convince hosts on other segments that its ethernet MAC address is the one associated with the IP address victim host, but by using IP filters, actual IP packet communication through the bridge can be prevented.

3 Bridging and IPsec

The filtering capabilities offered by the bridge allow its use as a transparent packet filtering firewall. As was the case with traditional firewalls however, filtering by itself is not sufficient in fulfilling network security concerns. Network layer encryption, typically in the form of IPsec, is seeing increasing use in protecting traffic between networks, hosts, and users. Thus, we decided to augment the filtering bridge with IPsec capabilities.

This section starts with a brief overview of the IPsec implementation in OpenBSD, then describes the two configurations where bridging and IPsec may be used together.

The first of these configurations, “virtual LAN,” is used to transparently and securely connect ethernet segments over a wide area network. This is achieved by encapsulating ethernet frames inside IPsec packets which are then transmitted to a remote bridge that removes the protection and forwards the frames to the local LAN.

The second configuration is what the standards call a “bump in the wire” (BITW) implementation [9], wherein a security gateway (the bridge) transparently implements IPsec on behalf of one or more “protected” hosts. This allows gradual introduction of IPsec in a network without changing the end host configuration or software. This configuration is also a common design feature of network security systems used by the military.

Perhaps more importantly, such a transparent IPsec gateway can be used to enforce security properties for communications between the protected (or supervised) hosts and the rest of the world. Packets traversing the gateway can be examined and, depending on system policy:

- They may be forwarded or dropped, similar to a packet filtering firewall.
- Outgoing packets may cause the security gateway to attempt to establish a security association (SA) with the destination host, pretending to be the originating host, if the packets are unencrypted. If the packets are already IPsec-protected, it could simply forward them (or, in our case, bridge them). Naturally, the security gateway may always opt to establish an SA

with the destination, regardless of the existing security properties of the packet stream.

- Similarly, for incoming packets, the gateway could establish a security association with the originator if the packet was received unencrypted and/or unauthenticated, again pretending to be the destination host.
- Finally, the bridge can intercept incoming IKE [5] packets that request negotiation with one of the protected hosts, and perform the negotiation as that host.

Thus, in the last three cases, the security gateway acts as a transparent network policy enforcer. A routing firewall can perform the same functions, however it would have to establish tunnel SAs between itself and the remote host on behalf of the protected host. It would do so using its own IP address, and so would need to “prove” its right to proxy-IPsec for the end host. While this is trivial for static configurations, where the identities and network addresses of the two peers are known a priori, the situation becomes more complicated when trying to do opportunistic encryption.

The two primary envisioned methods for host authentication are DNSSEC [3] and X.509 [2]. In the former case, the domain name servers can securely provide the public key associated with a host name or address. That key may then be used to authenticate the IKE peer. In the X.509 case, a Certification Authority (CA) infrastructure is assumed to provide the public key of an end host or user (the protocols for doing so in a large-scale network such as the Internet are less well-defined than DNS). Here, the IP address of the host associated with a key is embedded in an X.509 certificate. In either case however, it is not immediately clear (and currently undefined) how to express the right of a firewall to establish SAs on behalf of a host. While work has recently started in the IETF IP Security Policy (IPSP) Working Group, development and deployment of a protocol that would allow security gateway discovery is some years away.

3.1 OpenBSD IPsec

IPsec in the OpenBSD kernel is implemented as a pair of transport protocols [7, 8]. Incoming IPsec packets are switched to the appropriate IPsec protocol for processing by *ipv4_input()*, following the

usual packet processing path in the kernel (similar, for example, to TCP or UDP). Note that only packets destined for the local host are handled this way; IPsec packets that are passing through are simply forwarded if the host is configured to act as a router, otherwise they are dropped).

Outgoing packets are intercepted at *ip_output()*, where a check is made to see if IPsec processing is necessary. If so, the appropriate IPsec protocol output routines are called which encrypt/authenticate the packet, and then re-send it via *ip_output()* specifying that IPsec processing has already occurred (so as to avoid loops). Two methods are used to determine whether a packet needs to be IPsec-processed:

- If the packet originated from a local socket, it may have an attached Security Association (SA).
- If no such information is available, the source and destination addresses and ports from the packet are used to make a lookup in the kernel Security Policy Database (SPD). In OpenBSD, the SPD is implemented as a new protocol family in the radix tree, which is also used for routing entries. There are several benefits to using the radix tree:
 - Code reuse (and, thus, bug avoidance).
 - The radix tree internal representation is compact, allowing for large numbers of SPD entries without high memory requirements.
 - Lookup cost scales well with number of entries in the table.
 - Because a lookup returns the most specific result, it is easy to implement “backup” entries for packet classes (or, conversely, we can have special case handling of certain packet classes).

In both cases, the lookup provides enough information to locate the SA. Note that, on input, the packet itself contains enough information to locate the SA. The SA itself contains such information as the cryptographic algorithms and keys to be used, what optional features of the protocols are in use, various expiration timers, *etc.*

Both the SA and SPD tables may be populated either through manual keying utilities, or by some

automated key management daemon (like IKE [5] or Photuris [6]). The interface to the kernel for either of these methods is via the PF_KEY socket [14], which is in many ways similar to the BSD routing socket.

Both in input or output, if the necessary cryptographic material has not been negotiated with the remote IPsec endpoint (for example, when doing on-demand or “opportunistic” IPsec), it is possible to notify a key management daemon which will then negotiate and install the proper SA and SPD entries in the kernel.

We have also implemented the *enc* pseudo-interface. Input packets that are IPsec-processed are made to appear as if they were received from the *enc* interface, by changing the interface pointer in the mbuf header. Thus, an administrator can easily filter non-IPsec protected packets using any packet filtering package. Furthermore, utilities like *tcpdump* [13] can be used to view the intermediate products of IPsec processing, for debugging purposes (this only works if IPsec processing takes place in the local host).

A more extensive (if somewhat dated) overview of the OpenBSD IPsec architecture is given in [10].

This is the standard IPsec processing that is more or less common across different implementations (and even operating systems). Use of IPsec in conjunction with the bridge, especially in the “bump in the wire” scenario, requires somewhat different processing. We shall describe these changes and requirements in the next two subsections.

3.2 Virtual LANs

Given the way the bridging code operates, in particular with respect to member interfaces being added to and removed from the bridge, it was a simple observation that we could extend the role of the *enc* interface so that it could be used by the bridge. Accordingly, we modified the *enc* interface such that an incoming and an outgoing SA can be associated with it, through the standard *ifconfig* command². Currently, such SAs must be manually configured, via the *ipsecadm* utility.

²As an artifact of our implementation, more than one SAs can be associated with an *enc* interface.

The effect of these changes is that local area networks (LANs) may be bridged over a public network. All that is necessary is that each LAN contain an IPsec bridge connecting it to one or more other LANs. From the point of view of the bridge, the IPsec link is identical to an ethernet network, allowing for creation of arbitrary topologies. Layer-2 and Layer-3 filtering, spanning tree algorithms, *etc.* may all be used in the IPsec-bridged network with literally no changes.

The modifications needed to the *enc* and *bridge* interface code were minimal. For the bridge, the only changes necessary were to allow non-ethernet interfaces to be attached and initialized properly (for example, it is not necessary to switch an *enc* interface into promiscuous mode). In the *enc* code, the routine that handles transmission was augmented to pass all enqueued ethernet frames to IPsec for processing and further transmission, after encapsulating them in IP or IPv6. Note that no SPD lookup is necessary here, since the output SA to use is already known.

To support multiple bridge topologies on the same host, a configurable number of *enc* interfaces is created. This number may be set at kernel compile time. By convention, packets received on SAs that do not have an *enc* interface associated with them, are made to appear as if they arrived on the *enc0* interface. Furthermore, the *enc0* interface is not allowed to have any SAs associated with it, nor can it be attached to a bridge. Thus, packets on SAs that have an *enc* interface associated may be traced or filtered using that interface. For all other SAs, the *enc0* interface may be used.

Implementation of the Ethernet-over-IP protocol also proved straightforward, as the output side of the protocol and the first half its input processing are identical to IP-in-IP encapsulation. At the end of input processing, if its input interface is linked to a bridge, the packet is passed to the bridge input routine. If a frame is received over an IPsec SA, its input interface will be the *enc* interface associated with the SA (see Figures 1 and 2).

In all, less than 300 lines of additional code in IPsec and the bridge were necessary to implement the virtual LAN functionality.

When it comes to performance, the highest cost is, as might be expected, in the encryption. Figure 3 shows the cost of various algorithms (and combina-

tions thereof). Note that AH only performs authentication (the packets are unencrypted).

Note however that it is usually the case that the wide-area network link over which the virtual LAN is established is much slower than the member LANs, and slower than the times shown above. Thus, realistically, the performance is limited mainly by the interconnecting infrastructure. The filtering capabilities of the bridge (blocking multicast/broadcast and non-IP packets) can be of some value here in managing the volume of traffic sent over the encrypted links.

Virtual LAN (vLAN) functionality is offered by a number of bridges, albeit it is used to mean something different from what we have described; more specifically, vLANs are used to compartmentalize a physical network into a number of "isolated" LANs. The main goal is to decrease the traffic "noise" as seen by machines that do not need to process it (*e.g.*, a printer does not need to receive NFS packets; likewise, normal hosts on the subnet probably do not need to see the AppleTalk packets the printserver uses to submit jobs to the printer). A side effect of vLAN employment is a limited form of security from casual packet sniffing. Such vLANs do not provide the same features our encrypting bridge offers (and vice versa).

3.3 Bump In The Wire

As mentioned in section 3, the bridge can also be used as a transparent IPsec box, sitting in front of a host or network and IPsec-processing packets traversing it. This configuration is called "bump in the wire" (BITW) in the IPsec architecture. The encrypting bridge as described in the previous section can be used almost as-is when the protected hosts or networks are configured to only talk to one remote host (or security gateway): an incoming and outgoing SA pair can be associated with an *enc* interface as before, and IPsec processing is done along the same lines. However, rather than encapsulating ethernet frames inside IP packets (and then IPsec-processing these), we extract the IP packets from the ethernet frames and do IP-in-IP encapsulation instead. The administrator can specify which of the two types of encapsulation should be used simply by setting the appropriate interface flag using the *ifconfig* command.

Transform	Mbit/second
Software AH MD5	67.87
Software AH SHA1	47.79
Software ESP DES-MD5	19.56
Software ESP Blowfish-SHA1	23.61
Software ESP 3DES-SHA1	9.07
Hardware ESP DES-MD5	62.12
Hardware ESP 3DES-SHA1	63.12

Figure 3: Throughput of a TCP session over IPsec between two K6-3/550 boxes directly connected with 100Mbit/s ethernet. For the hardware numbers, we used a card with the Hi/Fn 7751 chip.

The SAs associated with the *enc* interface (which must be manually configured) can use the IP address of the bridge, or the IP address of the protected host. In the former case, the bridge exhibits the exact same characteristics as an encrypting gateway (packets sent to the remote host or gateway list the bridge's IP address as the source); in contrast to a gateway however, no configuration changes are necessary in the network or the protected host(s) when placing the bridge. Since SA configuration is manual, there are no issues with authentication during key establishment (as described in section 3).

When the SAs use the IP address of the protected host, the bridge is totally transparent to both the protected host and the destination host or gateway. There are two issues that need to be addressed in this configuration however:

- The IPsec standard specifies that IP fragments should not be IPsec processed in transport mode. That is, fragmentation must occur after IPsec processing has taken place, or tunnel mode (IP-in-IP encapsulation) must be used. Thus, the bridge must either use only tunnel mode IPsec, or reassemble all fragments received from the protected host, IPsec-process the re-constructed packet, then fragment the resulting packet. For performance and simplicity reasons, we decided to use the former approach. The disadvantage is that all IPsec-processed packets are 20 bytes larger (the size of the external IP header).
- Since the remote host is not aware of the encrypting bridge's existence, IPsec packets will be addressed to the protected host or network. Thus, we have to modify the bridge to recognize these addresses and process those IPsec packets. In fact, address recognition is unnecessary.

The bridge only has to watch for IPsec packets (transport protocol ESP or AH), and use the information in the packet to perform an SA lookup. If an SA is found, the packet is processed. Otherwise, it may be dropped or allowed through depending on the filtering configuration.

A hybrid SA configuration may be used (where the bridge uses its address in one direction, and the protected host's address in the other). Such a configuration does not seem to offer any substantial benefit however (and may in fact result in confusing the administrator).

3.4 Transparent Policy Enforcement

While the mechanism described in the previous subsection is useful in its own right, its usefulness dramatically expands when the bridge is modified such that it can automatically establish SAs on behalf of the protected hosts.

Our IPsec implementation already supports dynamic SA acquisition by notifying a key management daemon like *isakmpd* [4] using the PF_KEY interface. SA acquisition occurs in the following two cases:

- An application requests some security service on a socket, by using the *setsockopt()* system call, and no SAs appropriate for the traffic pattern or security level exist.
- The kernel decides that a packet needs to be IPsec-processed, but no appropriate SAs exist. The kernel reaches this decision by consulting the SPD (as described in section 3.1).

We can use the same mechanism inside the bridge to dynamically establish SAs: when an IP packet (rather, an ethernet frame containing an IP packet) reaches the bridge and is about to be “transmitted” over an *enc* interface, an SPD lookup is made. If an SA appropriate for this packet already exists, it is used. Otherwise, the packet is dropped and a notification is sent to the key management daemon to establish such an SA. The granularity of the SA may be configured by the administrator (the same SA may be used for all traffic between the protected and the remote host, or just the specific TCP connection may be protected). Future packets with the same characteristics as the original packet will make use of the newly-established SA. Fortunately, no changes to the SPD are necessary.

This mechanism may be used to perform automatic re-keying for the virtual LAN or the simple “bump in the wire” configurations described in the previous two subsections.

However, left to its own devices, key management will establish an SA using the IP address of the bridge (and thus end up being functionally equivalent to an encrypting gateway). To really hide the bridge from the remote host, the source address of the protected host must be used. Thus the key management daemon, *isakmpd*, has to operate in the “bridge mode,” wherein it asks the kernel to use a non-local IP address. This requires a minor change in the *bind()* system call code, to allow for socket binding to non-local addresses. To capture the responses, all UDP traffic to port 500 (the port used by the IKE protocol) is diverted to the bridge *isakmpd*. This is also necessary when remote hosts attempt to initiate an IKE exchange with a protected host. In both cases, *isakmpd* must be informed of and use the “local” address associated with the incoming packet. *isakmpd* also needs the “local” address so as to select the appropriate authentication information (e.g., secret DSA [15] or RSA [12] key when doing X.509 or DNSSEC authentication). The changes to this effect are fairly minimal.

Incoming IPsec packets are processed as described in the previous subsection. Other incoming packets may cause an SA acquisition, depending on the security policy set by the administrator. Again, *isakmpd* needs to be informed of what IP address to use as the source address during the exchange.

The combination of packet filtering through *ipf* and

SA-on-demand can be used effectively to enforce network security policy for the protected host(s). One particularly interesting configuration involves the bridge establishing SAs for unencrypted-only traffic; if end-hosts use IPsec or SSL for end-to-end packet security, the bridge simply forwards the packets. In another configuration, the bridge permits all packets through, but attempts to establish SAs for such communications and uses them if the remote hosts can do IPsec (also known as “opportunistic encryption”).

4 Implementation Status and Future Work

Currently, the bridge lacks support for the spanning tree protocol which is part of the IEEE 802.1D standard[16], so care must be taken to ensure that loops are not created in the network. The Layer-2 filter rule system should be extended to provide a general mechanism for filtering specific ethernet protocols. We also intend to extend the bridge to allow for other types of LAN bridging (FDDI, PPP, etc.).

With regards to dynamic SA establishment, all traffic that traverses the bridge configured in the manner described in section 3.4 causes SA acquisitions. This is both undesirable and can have severe performance implications. A mechanism for the administrator to specify which packet flows should require IPsec protection (and thus cause an SA acquisition) is necessary. We are currently working on this issue.

More work needs to be done with regards to the performance implications of frequent IKE negotiations, as might be the case when the bridge is protecting a large network. Hardening against denial of service attacks (by exploiting too-aggressive SA acquisition rules) is also high in our to-do list.

The filtering bridge can also provide a transition step for a “distributed firewall”-protected network, as described in [1]. It may also be used in conjunction with a distributed firewall to provide protection against low-level network attacks (those that a distributed firewall is not well-suited to counter), or to protect legacy systems that cannot be modified to support the required functionality. Very low-priced systems (motherboard, processor, small disk, two ethernet cards, moderate amount of memory) may be used in such a configuration; such systems may

also be used as “personal firewalls,” similar to various commercial products that have begun to make their appearance in the market recently.

5 Conclusions

We have given an overview of the OpenBSD bridge implementation, with our extensions for Layer-2 and Layer-3 filtering (at the ethernet and IP layer, respectively). For the latter, we used the existing kernel packet filter mechanism, *ipf*. We further presented our integration of bridging with IPsec to provide “virtual LAN” functionality, “bump-in-the-wire” support, and a transparent security policy enforcement box. This latter configuration is shown to offer significant flexibility to network administrators, as it can be used in various modes of operation to ensure traffic as well as host and network protection. Finally, we discussed the current implementation status and our plans for future work.

6 Acknowledgments

The bridge was originally developed as an undergraduate independent study at the University of North Carolina at Greensboro by Jason L. Wright with Dr. Suzanne Lea as an advisor. The code was contributed to the OpenBSD project and integrated into the source tree prior to the 2.5 release (May 1999).

The authors would also like to thank Theo de Raadt and Jonathan Smith for their suggestions and support during the course of this work. Theo de Raadt suggested many of the original concepts behind the filtering bridge and the virtual LAN. This work was partly sponsored by DARPA under grant F39502-99-1-0512-MOD P0001.

7 Availability

All the software described in the paper is available through the OpenBSD web page at:

<http://www.openbsd.org/>

8 Disclaimer

OpenBSD is based in Calgary, Canada. All individuals doing cryptography-related work do so outside countries that have limiting laws.

References

- [1] S. M. Bellovin. Distributed Firewalls. *login: magazine, special issue on security*, November 1999.
- [2] Consultation Committee. *X.509: The Directory Authentication Framework*. International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.
- [3] D. Eastlake and C. Kaufman. Dynamic Name Service and Security. Internet RFC 2065, January 1997.
- [4] Niklas Hallqvist and Angelos D. Keromytis. Implementing Internet Key Exchange (IKE). In *Proceedings of the Annual USENIX Technical Conference*, June 2000.
- [5] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments (Proposed Standard) 2409, Internet Engineering Task Force, November 1998.
- [6] P. Karn and W. Simpson. Photuris: Session-key management protocol. Request for Comments (Experimental) 2522, Internet Engineering Task Force, March 1999.
- [7] S. Kent and R. Atkinson. IP authentication header. Request for Comments (Proposed Standard) 2402, Internet Engineering Task Force, November 1998.
- [8] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). Request for Comments (Proposed Standard) 2406, Internet Engineering Task Force, November 1998.
- [9] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, November 1998.
- [10] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom) '97*, pages 1948 – 1952, November 1997.
- [11] Kirk McKusick, et. al. *The Design and Implementation of the 4.4BSD Operation System*. Addison Wesley, 1996.
- [12] RSA Laboratories. *PKCS #1: RSA Encryption Standard*, version 1.5 edition, 1993. November.

- [13] Steven McCanne and Van Jacobson. A BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of USENIX Winter Technical Conference*, pages 259–269, San Diego, California, January 1993. Usenix.
- [14] D. McDonald, C. Metz, and B. Phan. PF_KEY Key Management API, Version 2. Request for Comments (Informational) 2367, Internet Engineering Task Force, July 1998.
- [15] Digital Signature Standard, May 1994.
- [16] Internetworking Task Group of IEEE 802.1. Information technology – telecommunication and information exchange between systems – local and metropolitan area networks – common specifications – part 3: Media access control (mac) bridges. Technical Report ISO/IEC Final DIS 15802-3, IEEE P802.1D/D17, LAN MAN Standards Committee of the IEEE Computer Society, May 1998.

Safety Checking of Kernel Extensions

Craig Metz

Department of Computer Science

University of Virginia

Charlottesville, VA 22904

`cmetz@inner.net`

Abstract

There are many places in operating systems today where extending the running kernel with small and fast extensions is an interesting thing to do. For example, the Berkeley Packet Filter (BPF) allows code for a virtual machine to be uploaded into a running kernel and executed at packet reception, allowing fairly arbitrary filtering of packets before they cross the expensive kernel to user interface. Whatever mechanism is used needs to provide some reasonable guarantees about the safety of the resulting code, which makes this problem complex.

This paper describes a simple x86 bytecode verifier that is intended to be used to verify that a small program that is to be loaded obeys a reasonable safety policy. For program constructs that it is able to reason about, it can verify that code does not execute privileged instructions, only accesses known memory locations, and terminates. It cannot reason about arbitrary programs, but can reason about simple programs and developers that know the prover's limitations can write their code to be recognizable by the verifier.

The contribution of this work is to show that a very limited prover can operate on native machine code and can efficiently reason about a small but still interesting set of programs.

1 Introduction

The Berkeley Packet Filter (BPF) is a simple virtual machine that is commonly found in the kernels of BSD-like systems. It allows user processes (that may or may not be privileged) to install simple extensions into the running kernel that are called at well-defined points. These extensions provide a service in kernel space on behalf of the user process that installed them, and provide greater flexibility than a data-only configuration interface can. The virtual machine has a fairly high overhead due to both the instruction set emulation and the run-time access checking, but this is much lower

than the overhead of crossing into the user process.

Increasing network speeds and interest in applying this approach to other problems have made emulation overhead an issue. Also an issue is that the BPF program must be written in a special assembly language; no high-level language compilers currently exist that target the BPF machine. Native machine code has neither problem; it executes at full machine speed and can be targeted by whatever compilers are available. Many systems have a facility for extending the running kernel using native instructions (loadable kernel modules). Typically, these extensions are installed by a privileged process, are not subject to any sort of verification at installation time, and have the same run-time privileges as the rest of the kernel (in particular, there is usually almost no access checking). Such facilities are inappropriate for use by unprivileged processes, and represent serious risks even when restricted to privileged processes. In particular, the extension code can put the entire system in a bad state, either due to deliberate actions or unintended flaws.

A possible solution to the problem is to statically analyze program code to determine whether it obeys certain restrictions and safety policies. BPF already does this in a very limited way. The `bpf_validate()` function is called before code is installed into the running system, and checks three safety properties [1]:

1. "Check that jumps are forward, and within the code block."
2. "Check that memory operations use valid addresses" (within the VM)
3. "Check for constant division by 0"

At run-time, the BPF virtual machine provides memory access checking (by only allowing access through well-known pointers and bounds checking of offsets), provides no unsafe instructions, and guarantees that the C calling convention will be obeyed (because the virtual machine itself is implemented as a C function). Still, the run-time checking creates overhead.

A similar approach is used by the Java Virtual Machine (JVM), which uses a bytecode verifier to check for memory and type safety as code is loaded into the system[2]. Again, some safety is provided by the emulated machine, which provides no truly unsafe instructions and guarantees that the calling convention will be obeyed. However, the JVM approach alleviates the need for run-time bounds checking of memory accesses by moving that overhead to a load-time static analysis. While this improves run-time performance, there is still significant overhead due to the emulation of the JVM's instruction set.

In theory, the instructions present in the BPF virtual machine's instruction set and the JVM's instruction set are functionally equivalent to instructions (or short sequences of instructions) in native machine instructions. If it is possible to reason about the BPF or JVM instructions statically, it may be possible to do the same with a constrained version of a real machine's instruction set. This is an active area of research typically for use with micro-kernels, where fast and safe kernel extensions are more commonly needed. A particularly interesting research solution is Proof Carrying Code (PCC) [3], in which native code is statically analyzed, an attempt is made to generate a proof that the code obeys certain safety properties, and the proof and code are later verified by the kernel before accepting the code as an extension.

If a proof can be found and verified, then the code is known to have the safety properties and run-time checks are not needed [4]. If a proof cannot be found, this does not necessarily mean that the code is unsafe – as a nontrivial program property, this is an undecidable problem[5], but it is always a safe default action to consider code to be unsafe. If a programmer has the prover in hand and knows what the safety policy is, development can be done iteratively where the programmer adjusts the code in equivalent ways until it passes the prover; while the prover might not be able to reason about arbitrary programs, it is usually possible for a programmer to find a functionally similar program that it can reason about.

The main problem with the PCC approach is that the prover is too heavyweight, yet is still very limited in its ability to actually prove properties of programs. In one PCC implementation, the prover is a general-purpose theorem prover, which is very large and slow, but can attempt to prove rather arbitrary properties of programs. In practice, this approach becomes far less likely to successfully generate a proof as program complexity increases. In another, the prover is built into a special type-safe C compiler; most of the overhead of the prover is shared with the compiler and proofs are always generated for valid programs in the language (the property proved is type safety and the language

is a type-safe language, so one is always possible), but now a specific compiler must be used. Because the proof generation step is so heavyweight, the PCC approach separates proof generation and proof verification and makes only the latter a trusted component.

A promising direction for a solution to this problem is to start with the PCC approach and to make the prover sufficiently fast and lightweight that the proof generation can be a trusted component (eliminating the need for an intermediate representation of the proof and a proof verifier). This is done by greatly constraining what programs are allowed to do; if programs are simple enough, reasoning about them becomes simple also. In particular, extensions are currently required to be stateless and return a simple integer result. This is still interesting for many problems such as packet filtering, and allows the extension to be terminated by the system without having to worry about cleanup. Conditions like division by zero are already checked at run-time “for free” by the hardware and global state can be recovered from rather painlessly, so there is little value in a relatively expensive static analysis to decide whether the condition is possible. In contrast, safety of memory accesses is expensive to do with the hardware (due to the time required to switch to more restrictive page tables), and backing out a write to a random memory location is painful if not effectively impossible, so statically checking for this kind of safety is valuable.

2 Prototype Prover

2.1 General Approach

The prover that was constructed takes native machine code and executes it using a simple simulator. The implementation currently only supports the x86 instruction set, though there is nothing in this approach that would prevent an implementation for another Instruction Set Architecture (ISA) (and most ISAs will actually be far easier to build an implementation for, but the popularity of the x86 instruction set motivated its use in this prototype). Privileged machine instructions and machine instructions that would not be output by a normal linear-mode x86 compiler (e.g., media instructions and instructions using segmentation) are not allowed in extensions. Floating point is also not allowed because hardware floating point instruction support may not be present on the machine and is not automatically emulated for code inside many kernels. Known values are tracked through the simulated execution of the program using a simple linear representation:

$$value = base + a \times x + b, x = 0..x_{max}$$

Where *base* is either 0 or an abstract unknown base variable whose concrete value will be supplied by the runtime environment (such as the initial value of the stack pointer, or the pointer to the function's input structure). Preconditions, such as the input values, descriptions of the accessible memory objects (alignment, size, and permissions), and the C calling convention's stack layout, are set up with their values. All other memory and register values in the system are set to "undefined," and the simulation is run. At all return points, postconditions such as register and stack restoration are checked.

Memory accesses are required to be to known values. Given a known value as above and memory objects in a similar representation, it is possible to check whether a memory access will always be in a defined region for which the program has permission to read or write. Known values are tracked on write, so that register spills to the stack do not lose information.

The simplifying assumption that makes the simulator function is that it is always safe to declare a value to be unknown if a known value cannot be easily computed. For example, most bitwise logical instructions yield a result of unknown because the simulator cannot create a linear function representation of their output given a linear function representation of their input. The main exception to this is bitwise AND, which is treated as constraining the linear function's range. This works in practice because most memory accesses are done through linear functions (scale-index-base, as seen in the x86 ISA), so the linear values tracked by the simulator contain enough information to reason about most memory addresses, while the other values used by a program aren't usually important for memory access safety.

The end result is a prover that can take short binaries compiled with `gcc` and verify their safety. The verified code can then be installed into a running kernel and used for small extension functions in the same way that the BPF virtual machine is used. These functions can be called as C functions through a front-end function, and are guaranteed to return and not to write except where explicitly permitted.

2.2 Implementation Walk-Through

In order to make discussion of the implementation more concrete, Figures 1-4 give an example of a short real-world filter program that was used extensively in the development of the prototype prover. First, a short libpcap filter program shown in Figure 1 was used as a starting point. It was chosen to be a short example of how BPF is typically used. The `tcpdump` program was used as a front-end to libpcap's internal compiler and optimizer, which generated the BPF program shown in Figure 2.

ip host 127.0.0.1 and udp port 42

Figure 1: Original libpcap filter program syntax

```
(000) ld      [0]
(001) jeq     #0x2000000      jt 2      jf 16
(002) ld      [16]
(003) jeq     #0x7f000001     jt 6      jf 4
(004) ld      [20]
(005) jeq     #0x7f000001     jt 6      jf 16
(006) ldb     [13]
(007) jeq     #0x11          jt 8      jf 16
(008) ldh     [10]
(009) jset    #0x1fff        jt 16     jf 10
(010) ldx     4*([4]&0xf)
(011) ldh     [x + 4]
(012) jeq     #0x2a          jt 15     jf 13
(013) ldh     [x + 6]
(014) jeq     #0x2a          jt 15     jf 16
(015) ret     #1576
(016) ret     #0
```

Figure 2: BPF program resulting from compilation

Then the C program shown in Figure 3 was constructed by hand, attempting to be as faithful to the contents of the BPF program as possible. However, the C implementation uses abstractions for protocol headers and address information for the sake of readability and to show that programs that might be written by a real programmer compile down to analyzable code. The C program adds some run-time bounds checking that is not present in the BPF instructions – the BPF virtual machine does these checks implicitly, while a native program must do them explicitly.

Finally, the C program was compiled with the GNU C compiler into a native x86 program whose assembly listing is shown in Figure 4.

The first thing that the prover does is load the code of the program to be proven into a memory buffer. This requires the program to parse the i386 ELF binary object format, which is currently done in a reasonable but minimal fashion. Only the `.text` segment is loaded, which is assumed to contain exactly one C function, and no relocation is performed. The program loader is currently linked into the same binary as the prover, but is otherwise decoupled from it. In practice, the loader would be separated from the prover and would be an untrusted component (for example, a part of the user process).

Next, the prover initializes preconditions. The register pre- and post-conditions are listed in Figure 5, and are generic to the x86 C calling convention. The first four general purpose registers are initially set to

```

#include <sys/types.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>

struct bpf_preamble {
    uint32_t a;
};

int f(caddr_t p, unsigned int len)
{
    struct bpf_preamble *bpf_preamble;
    struct ip *ip;
    struct udphdr *udp;

    if (len < sizeof(struct bpf_preamble))
        goto fail;

    bpf_preamble = (struct bpf_preamble *)p;
    if (bpf_preamble->a != PF_INET)
        goto fail;

    if (len < sizeof(struct bpf_preamble)
        + sizeof(struct ip))
        goto fail;

    ip = (struct ip *) (p + sizeof(struct bpf_preamble));

    if (ip->ip_hl < 5)
        goto fail;
    if ((ip->ip_src.s_addr != htonl(INADDR_LOOPBACK)) &&
        (ip->ip_dst.s_addr != htonl(INADDR_LOOPBACK)))
        goto fail;
    if (ip->ip_off & IP_OFFMASK)
        goto fail;

    if (len < sizeof(struct bpf_preamble) + ip->ip_hl * sizeof(uint32_t) +
        sizeof(struct udphdr))
        goto fail;

    udp = (struct udphdr *) (p + sizeof(struct bpf_preamble) +
        ip->ip_hl * sizeof(uint32_t));
    if ((udp->uh_sport != htons(42)) &&
        (udp->uh_dport != htons(42)))
        goto fail;

    return 1576;

fail:
    return 0;
}

```

Figure 3: Equivalent C source code

```

.text
    .align 4
.globl f
    .type    f,@function
f:
    pushl %ebp
    movl %esp,%ebp
    pushl %esi
    movl 8(%ebp),%ecx
    movl 12(%ebp),%esi
    cmpl $3,%esi
    jbe .L8
    cmpl $2,(%ecx)
    jne .L8
    cmpl $23,%esi
    jbe .L8
    leal 4(%ecx),%edx
    movzbl 4(%ecx),%eax
    andl $15,%eax
    cmpl $4,%eax
    jle .L8
    cmpl $16777343,12(%edx)
    je .L18
    cmpl $16777343,16(%edx)
    jne .L8

.L18:
    testl $8191,6(%edx)
    jne .L8
    leal 0(,%eax,4),%edx
    leal 12(%edx),%eax
    cmpl %eax,%esi
    jb .L8
    leal 4(%ecx,%edx),%eax
    cmpw $10752,(%eax)
    je .L25
    cmpw $10752,2(%eax)
    jne .L8

.L25:
    movl $1576,%eax
    jmp .L26
    .p2align 4,,7

.L8:
    xorl %eax,%eax

.L26:
    popl %esi
    movl %ebp,%esp
    popl %ebp
    ret

.Lfe1:
    .size    f,.Lfe1-f

```

Figure 4: x86 program resulting from compilation (GCC 2.95.2, -O6)

“undefined,” which is a special value in the prover that indicates that these cannot be read from until changed to a defined value (to prevent unknown or unintended values from being available to the program). The last four general purpose registers are initially set to abstract variables representing their initial value. This allows the program to generate values as offsets from their initial values (such as stack pointer relative addresses, which need to be resolved to an address relative to the initial value of the stack pointer) and it also allows the prover to check as a postcondition that the initial value has been faithfully restored when the program returns.

The memory preconditions are listed in Figure 6, and are specific to the problem of a BPF filter function with the function signature seen in Figure 3. Different functions will require different memory preconditions. The current prover implementation allows these to be changed easily. Memory words are defined to contain each input parameter, and the input packet buffer that is passed to the program is also defined. The input packet buffer is currently defined as a fixed-length buffer in order to greatly simplify the prover’s operation; it is much easier to reason about falling within a known bound than an unknown bound (though this would be a useful feature to add in the future). At run-time, the caller would need to copy packets into a

Register	Precondition	Postcondition
%eax - %edx	(undefined)	n/c
%esp	sp_0	sp_0
%ebp	bp_0	bp_0
%esi	si_0	si_0
%edi	di_0	di_0

Figure 5: Register Pre/postconditions for the x86 C Calling Convention

buffer of 8192 bytes in order for this to be safe. The input parameters are all treated as read-only because there is no legitimate need for them to be modifiable. In order to have some available scratch space, twelve words of stack space is set aside for read/write local variables. This scratch space is easily increased by defining more such words in the prover, but arbitrary amounts of temporary space are not supported. There are no memory postconditions; values that are read-only are never writable and thus need no postcondition checking, and values that are read-write are considered mutable.

Name	Base Variable	Offset	Size	Contents	Permissions
arg 0: <code>caddr_t p</code> at 4(%esp)	<code>sp0</code>	+4	4	<code>p</code>	read
arg 1: <code>unsigned int len</code> at 8(%esp)	<code>sp0</code>	+8	4	<code>len</code>	read
locals 0..11	<code>sp0</code>	-4..-52	4	(undefined)	read/write
Input packet (<code>p</code>)	<code>p</code>	0	8192	(unknown)	read

Figure 6: Memory Preconditions for a BPF-like Filter Program

2.3 Limited x86 Simulation in the Prover

The core of the prover implements a very limited simulation of an x86 processor. The intent of this simulation is not to run the program or yield results, but to quickly attempt reason about the instructions and values used in a program and to attempt to determine whether memory accesses are safe.

A surprisingly complex part of this process is decoding the x86 instructions into the actual operation and operands. Figure 7 shows the general form of a x86 instruction. The x86 is a CISC architecture and earns the designation “complex.” There are multiple opcodes for the same actual instruction, and the operands are determined by the opcode in ways that are frequently not well patterned. The instructions appear to be designed for a table-driven instruction decoder. For performance reasons, actual x86 chips probably implement this as optimized combinational logic, but development tools I looked at consistently used the table-driven decoding approach.

Appendix A of Intel’s ISA reference[6] contains “opcode maps,” which are tables of the mapping between opcode byte values and the instructions and operands they represent. Tools for the x86 instruction set commonly adapt these tables to drive their decoding of the instruction set, so this approach was chosen for use in the prover. The general structure of the tables works well for an implementation. There were many errors in the tables provided in Intel’s documentation; in most cases, these errors were obvious when read, but there were a few cases that were more subtle and caused problems that were found during debugging of the prover. I submitted the first few corrections to Intel’s developer support group and have yet to receive any response. Given that many of these errors are in parts of the tables that have not changed in over twenty years, this is unfortunate.

Figure 8 lists the subset of the Intel operand codes that are currently supported by the prover. Operands relating to segmentation, floating point, media instructions, and privileged or machine control instructions are all unsupported because the instructions that might use them are unsupported. Also unsupported are operand types not generated by `gcc`. These operand codes are used directly in the decode tables of

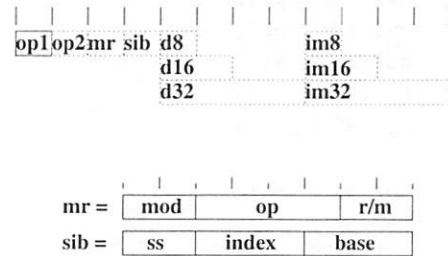


Figure 7: General Form of a x86 Instruction

the prover, along with a function pointer that calls a function that actually implements the operation specified.

An example of the resulting decode structure is the entry for opcode 0x89:

```
{ insn_mov, OP_Ev, OP_Gv, OP_NONE, FL_NONE },
```

This structure indicates that the actual operation is performed by the function `insn_mov()`, that its first parameter is a variable-length (that is, determined by the current word size mode) general purpose register, that its second parameter is a variable-length general purpose register or memory location, that this function has no third parameter (which most don’t), and that this operation does not affect the `%eflags` register.

A set of important special cases are bytecodes that are not opcodes at all, and instead are prefixes that modify the following bytecodes. Most of these fall into categories of instructions that are not allowed (e.g., segment prefixes, address-size prefix, and the `lock` prefix), but two that need to be supported are the word-size prefix (0x66) and the two-byte opcode prefix (0x0f). Both are handled as if they were no-operand opcodes, and the functions that implement them return special values that are accumulated into an internal set of flags in the instruction decoder. These flags are accumulated and affect decoding of subsequent bytes until an “ordinary” instruction completion, at which point they are reset. One flag, used to implement the two-byte opcodes, switches the table used to decode opcode bytes. The other flag, used to implement the word-size prefix, causes a temporary operand structure to be created for the current instruction in which variable-length operand specifications are replaced with 16-bit length operand speci-

Symbol(s)	Description
Ev/Ew/Eb	G.P. register, variable/16-bit/8-bit length
Gv/Gw/Gb	G.P. register or memory, variable/16-bit/8-bit length
Iv/Id/Iw/Ib	Immediate, variable/32-bit/16-bit/8-bit length
eAX/AL/AH	%eax, variable length, lowest 8 bits, next 8 bits
eCX/CL/CH	%ecx, variable length, lowest 8 bits, next 8 bits
eDX/DL/DH	%edx, variable length, lowest 8 bits, next 8 bits
eBX/BL/BH	%ebx, variable length, lowest 8 bits, next 8 bits
eSP/eBP	%esp/%ebp, variable length
eSI/eDI	%esi/%edi, variable length
CONST	constant operand (based on opcode) (non-Intel symbol)

Figure 8: Operand Types Currently Implemented

cations (thus changing them from 32-bit operands to 16-bit operands).

Instruction execution is simulated through short functions. These functions are not meant to model the actual execution of instructions. They are meant to track whether their results are predictable or not, and, if so, attempt to predict or put a range on the output values. So, for example, the `mov` instruction simply copies its first operand to its second; its second operand therefore has the same predictability properties as its first. In contrast, the `or` instruction takes two values and does a bitwise operation on them. If the two values are exactly known, an exactly known result can be generated; otherwise, the result of a bitwise or operation on at least one unknown quantity with a known quantity is difficult to reason about, and is simply considered to yield an unknown result. This is not a significant problem in practice because bitwise operations are not commonly used to generate addresses, and the prover is primarily interested in tracking values that are used as addresses.

One particularly tricky problem is conditional branch instructions. The prover was originally intended to use a reasonably sophisticated technique of tracking where result flags values came from and using some of those sources together with conditional branches to constrain values. For example, in order to implement reasoning about variable-length input blocks, the prover needs to be able to look at a comparison with the length variable and to separate what code is executed if the length is greater than or less than the value compared against. Otherwise, the code could be correctly checking for length and not doing accesses beyond the buffer, but the prover wouldn't be able to determine that. A simpler technique turns out to be surprisingly successful; all conditional branches are treated as a nondeterministic branch, and both forks are checked independently. Combined with a large fixed input buffer length, the inability to reason about values is not a problem. This also captures the

critical property that run-time conditionals are typically not certain in advance (otherwise there would be no need for such a conditional), but that both sides of the branch must be to code that will do something safe.

Levels of nondeterminism and the total number of instructions that can be executed in a path are bounded in the prover. This is not so much designed to bound the program (though that is a consequence) as it is to bound the run-time of the prover. The intended applications of this prover are places where speed and small memory usage are important, and must be rather tightly bounded. The current prover allows up to 32 levels of nondeterminism and up to 128 instructions per path. These limits allow short and simple programs to be proven, which is the intent of the prover, but are not enough to allow complex programs to be proven safe, which probably would be outside the capacity of the prover anyway. These limits are very easily increased at compile-time if needed. Note that nondeterministic branches currently cannot rejoin; they become separate from every other branch until they terminate.

2.4 Preliminary Results

Preliminary results show that the extension code is sped up by an order of magnitude by being written in optimized native code rather than emulated BPF code. For the example used earlier, described in Figures 1-4, executed on a 333MHz Celeron CPU and entirely in-cache, the native code took an average of $0.130\mu s$, while the BPF code took an average of $1.328\mu s$. This is consistent with the PCC project's results and with this project's expectations. This is the recurring cost and is the cost that executes synchronously with the performance-critical part of the system (packet reception from the network in real-time), so an order of magnitude speedup is very helpful.

The more important result for this project is the

performance of the prover itself. Related works, such as PCC, did not appear to make available solid information about the performance of their provers, but consistently indicated that their speed and memory consumption were fairly high (with the justification that that was a one-time cost). In order to be practical for inclusion into an operating system's kernel, the run-time and memory requirements for a prover must still be reasonably small. On the example used, the prover developed for this project took an average of 938 μ s of execution time, and required 29,964 bytes of heap space above 17,152 bytes of program (with a small amount of stack space also required). This is for a rough, un-optimized implementation. All of those requirements could probably be decreased linearly, but not by an order of magnitude. Implementing more complex reasoning in the prover might increase those costs.

3 Conclusions

This technique provides an alternative to BPF for packet filtering at gigabit speeds. In this application, an order of magnitude run-time performance increase is very helpful because receiving the packets is already pushing the limits of what common x86 hardware can do. With some straightforward modifications, the commonly used packet capture library libpcap can be made to generate native machine code rather than BPF code, which can then be passed to the kernel for verification and installation. Such an arrangement would allow many common packet-capture programs, such as tcpdump, ethereal, and the ISC DHCP server, to take advantage of this performance increase simply by linking with the new library. Alternately, libpcap could be completely bypassed and a compiler could be used to generate an optimized binary for installation in the kernel, but this would require applications to change.

The single greatest danger of this approach is that the prover is a trusted component and has not been developed using high-assurance software techniques. If the prover were to be flawed in such a way that an unsafe program could be loaded as an extension, it could circumvent the security of the entire system. The prover's implementation needs to be subjected to serious scrutiny before it could be deployed for use by unprivileged users. A reasonable strategy would be to initially constrain it to use by privileged users only.

Another problem is that the linear representation of values is done as the sum of several machine integers, each of which has as much precision as the sum will have in the running program. It may be possible to use this difference in precision to subvert the prover. A work-around that is currently employed is to constrain

the range of values allowed in each field of a known value, so that sign reversals are not possible as long as the base values are not close to the beginning or the end of the number space.

For the immediately intended application, static verification appears to be viable and the prover implemented for this project appears to be a successful proof of concept. This technique is probably also applicable to other problems, but care must be taken to address the generality and trust issues of building a practical prover.

There has been a good bit of similar work to this, both on verification of synthetic machine codes that make properties easier to reason about at a higher run-time cost and on extensive verification of real machine codes at a high proof cost. The expected contribution of this work is to show that a very limited prover can operate on native machine code (to get the best performance for the code that's executing) and can efficiently reason about a very small but still interesting set of programs.

4 Acknowledgments

This work started out as a project for a special topics seminar on microprocessor architectures taught by Kevin Skadron of the University of Virginia.

Niels Provos, Kevin Skadron, and Chris Telfer reviewed early drafts of this paper and provided valuable feedback.

References

- [1] U. C. Berkeley CSRG. bpf_filter.c. From 4.4BSD-Lite.
- [2] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.
- [3] George C. Necula and Peter Lee. Proof-Carrying Code. *CMU CS Tech Report CMU-CS-96-165*, September 1996.
- [4] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time checking. *Proceedings of the USENIX Second Symposium on Operating Systems Design and Implementation*, October 1996.
- [5] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, Boston, 1997.
- [6] Intel Corp. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference. 1999.

An Operating System in Java for the Lego Mindstorms RCX Microcontroller

Pekka Nikander

Helsinki University of Technology
Pekka.Nikander@hut.fi
<http://www.tcm.hut.fi/~pnr/rcx/>

Abstract

The Lego Mindstorms is a Lego bricks based robotics toy series produced by the Lego Group, based on the ideas developed at the Massachusetts Institute of Technology in the Programmable Brick project. The heart of a Lego robot, the RCX microcontroller, hosts a Hitachi H8 microcontroller with 28 kilobytes of memory available for downloadable firmware and applications. In addition to the GUI based programming environment provided by Lego, a number of alternative programming environments have been developed for the RCX. However, these alternative programming environments are written in C, tightly bound to the hardware, and provide only relatively low level services. The strong hardware dependency makes it hard to debug programs; in practice, a hardware simulator is needed, and such a simulator does not yet exist in an open source form.

In this paper we present a new type of operating system and new programming environments for the Lego RCX brick. The operating system is written almost completely in Java, and currently provides runtime support for Java, C and C++ programs. In the case of Java applications, simulation and debugging is relatively simple as it can be performed on a standard Java Virtual Machine with just a small hardware simulation package.

1 Introduction

Introduction of an affordable robotics development kit in the form of the Lego Mindstorms Robotics Invention System (RIS) [1], and its predecessor, the MIT programmable brick [2], has fostered a number of very different efforts for both teaching robotics and experimenting with non-traditional applications of robotic equipment. These approaches include, for example, the idea of using a hoard of Lego robots to collectively perform a larger task.

A Lego Mindstorms Robot consists of a programmable Lego brick, called the RCX, which contains three sensor inputs, three actuator outputs, four user buttons, a simple LCD display, an IR transceiver, and a Hitachi H8 microcontroller with 32 kilobytes of RAM, 4 kilobytes of which is used for interrupt vectors and other low level data. Normally, the RAM is used to host a firmware program, provided by Lego, which is used to interpret the actual user program. The user program is represented in the form of byte code [3]. The byte code itself, in turn, is generated on a Windows PC running a graphical programming environment, which is tightly bound to the Windows operating system. Currently, the byte code offers fairly limited view to the power of the RCX; for example, it allows only 32 variables to be used.

Since the two programming environments [4][5] provided by the Lego group are severely limited in their ability to fully utilize the computational power of the RCX brick, a number of independent programming environments have been developed for the RCX, including Not Quite C (NQC) [6], LegOS [7], and *librcx*, a minimal runtime library [8]. The latter two of these use the GNU C Compiler, part of the GNU Compiler Collection (GCC) [9], to compile C source code to the machine code of the Hitachi H8 processor.

In this paper, we present *RCX Java Operating System*, which is an experimental operating system for the RCX microcontroller. To compile our OS, we have used a modified GNU Java compiler, GCJ, from the GCC suite of compilers. The GCJ compiler compiles Java source code and byte code into the native code of the target machine. We used a cross compiler running on a FreeBSD PC with the Hitachi H8 as the target environment. In contrast with the other existing Java runtime environments, ours is almost completely written in Java, which was possible since all the Java code is being compiled into native code. In the project, getting the compiler and the runtime to function together was one of the

most interesting tasks. Furthermore, a number of interesting tricks were needed to represent non-object data structures in a beautiful way in Java code. Some of the design choices are explained in Sections 3 and 6.

In addition to being a neat way of making it possible to use Java to write programs for the RCX, our approach has a number of potential benefits. First, because the operating system is written in Java, it is fairly independent of the hardware, making it relatively easy to port to other microcontroller based systems such as the uClinux SIMM [10]. Second, as the operating system is written in an object oriented language, it is possible to extend the operating system using the object paradigm. Third, in Java the thread and synchronization models are tightly integrated into the language, making it natural to use threads when developing applications for the RCX. And fourth, the compiler and runtime system can be extended to support additional Java based technologies such as Jini [11]. Some of these aspects are further explored later in this paper, while others are left for future work.

The rest of this paper is organized as follows. In Section 2 we describe the hardware and ROM structure of the RCX microcontroller in more detail. Next, in Section 3, we briefly outline the structure of the GCC compilers, focusing on the GCJ native Java compiler, and describe the modifications we have made to it. Section 4 describes the minimal Java runtime that we developed to support Java based native code on the RCX, and in Section 5 we explain the RCX specific operating system services implemented as well as the interface from the Java environment both to the routines implemented in the RCX ROM and to the actual hardware. Finally, Section 6 summarizes the benefits and lessons learned so far, while Section 7 outlines some possibilities for future work.

2 Lego RCX microcontroller

The Lego RCX (see Figure 1) is a large Lego brick hosting a battery case, a Hitachi H8 microcontroller, an IR transceiver, a simple LCD panel, a few control buttons, sensor and actuator connectors shaped into the form of Lego brick connectors, and some auxiliary circuitry.

The standard programming environments provided by Lego [4][5] allow only a very limited access to the resources of the microcontroller. Basically, they are aimed to enable high school students to apply their knowledge of using Lego blocks in building physical structures on the domain of building logical program structures that

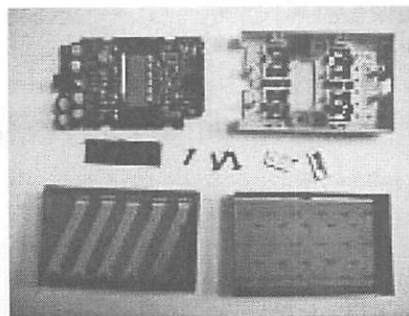


Figure 1: The Lego RCX, opened [12]

control the robot. However, our goal was to enable full access to the microcontroller resources. Therefore, only the low level programmer's view of the RCX is described. The following information is largely based on the reverse engineering work performed by Kekoa Proudfoot and others [12].

When programming at the machine language level (either using an assembler or through a high level language), a programmer has direct access both to the hardware and to the ROM routines. Hitachi H8 uses memory mapped I/O, and the actual hardware ports are mapped to the highest part of the memory, at a so called eight bit area. The I/O port map, as used for RCX, is partially illustrated in Table 1. [12]

Memory Range	Function
F000 - F0FF	Motor control
FFB7	IR transceiver range, button input
FFBA	IR control, external RAM power save mode
FFBB	Sensor power, timer, LCD I/O
FFBE	Sensor input, button input
FFC3	Serial/Timer control
FFE0 - FFE4	Sensor A/D input

Table 1: Some RCX I/O addresses

The ROM contains a fairly large number of routines, and it is beyond the scope of this paper to describe them all. However, the routines include the following functionality:

- Initialization functions and a simple main loop
- Default interrupt handlers
- Memory move, copy, clear, etc. auxiliary functions
- Battery power management
- Sensor I/O taking care of interpreting raw data
- Motor control
- LCD and sound output
- IR transceiver I/O

The firmware code, either as loaded from internal storage or over IR, is stored in a memory area starting at hex 8000. The default firmware is meant to interpret user programs expressed in the form of byte code [3]. Since we do not use the byte code in any way, the structure and functionality of the default firmware are beyond the scope of this paper.

3 GNU Java compiler and libgcj runtime

The GNU Compiler Collection (GCC) is a suite of compilers, based on the original GNU C Compiler architecture as created by Richard Stallman and others, including C, Fortran, C++, Objective C, and Java compilers. These compilers share the same basic structure and a common set of back ends, including a back end for the Hitachi H8 series of processors. The C and C++ versions of these compilers had earlier been adapted for the RCX environment by Markus L. Noga and others [7] [8].

Utilizing the C and even C++ compilers of the GCC suite is pretty straightforward, even for an embedded stand-alone environment such as the RCX. Basically, the C compiler requires little runtime support while the C++ runtime support is relatively modest. However, the runtime support required by the GNU Java compiler, GCJ, is both much more complex and currently less mature than its C and C++ counterparts. Furthermore, the Java runtime support is provided as a separate package, called `libgcj`, and its connections with the actual compiler are largely undocumented.

When we started our work with the system, a new version of GCC, GCC 2.95.1, and a corresponding version of the runtime, `libgcj` 2.95.1, were just released. However, in that version the compiler had a number of immaturities and restrictions, some of which have later been relieved while others have not been. In our work, we have attempted to generalize away some of the immature features and restrictions, basically aiming for a compiler that would be more runtime independent.

3.1 Restrictions in GCJ 2.95.2 and -current

In order to facilitate understanding of the current status of the compiler and the way the compiler restrictions made our work a little bit harder, a brief outline of the GCC compiler structure and the definition of the Java programming language is needed. They are presented next.

GNU compiler structure. The compilers in the GCC suite are structured around a three pass architecture. First, a file to be compiled is read in, along with any explicitly or implicitly included files. While parsing, the compiler front end forms recursive data structures, called *trees*, of any global declarations encountered. In the case of Java, all classes and most methods and fields are considered global; a Java compilation unit may only contain classes belonging to a single package, and classes within a package have access to all non-private fields and methods of each other.

Intermixed with the first pass, whenever the parsing of a compilable entity such as a method or a class is finished, the second pass creates an intermediate RTL representation of the entity. RTL, short for *Register Transfer Language*, is a kind of an abstract machine language.

In the third pass, a compiler back end optimizes the RTL representations through a number of passes into assembler language. Target specific transformations are taken into consideration and used by the optimization routines.

While building the Java runtime we experienced a number of times the relative newness of the Java front end, and to a lesser degree the fact that the COFF (Common Object File Format [14]) back end had apparently not been used before with the Java front end. However, the back end problems were basically inability to handle the names of Java specific data types, i.e., arrays, as first class objects. More specifically, the back end needed a fix to mangle any symbols containing brackets.

The Java front end. The front end restrictions were more severe but also more subtle. To understand these, we have to dig deeper into the structure of the Java front end.

In contrast to the C programming language, which was designed to be independent from any runtime library, the Java language specification explicitly defines a number of runtime classes that belong to the `java.lang` package. Of these, especially `Object`, `Class`, and `Throwable` are fundamental. Additionally, the default semantics of a number of runtime checks expect certain subclasses of the `Throwable` classes, e.g., `NullPointerException`, which is thrown whenever a null reference is followed. Furthermore, if a native compiler is to provide information needed by the Java reflection API (the `java.lang.reflect` package), the compiler needs to supply information about the fields and methods of classes.

Field name	Field type and contents
vtable	pointer to a table of function pointers; initialized to point to the dispatch table of the class
sync_info	void pointer; initialized to null

Table 2: Instance fields silently inserted to the Object class by the unmodified GCJ

The GCJ compiler addresses these needs by handling a number of runtime classes specially, including Object, Class, Throwable, Error, Exception, and Thread. Of these, the compiler treated the classes Object and Class significantly differently from others, thereby creating unnecessary limitations for the runtime. More specifically, the out-of-the-box GCJ silently inserts a number of fields (see Tables 2, above, and 4, on the next page) into these classes, and does not allow any new fields to be defined in the corresponding Java source code. Furthermore, even if some of the silently added fields can be accessed by Java code to be compiled, classes containing such references cannot be compiled with any other Java compiler. Finally, some of the internally generated fields have types that cannot be represented in Java. The restriction of not allowing any other fields to be declared, along with the other two above mentioned features, made it relatively hard to build the runtime in Java. Lifting the restriction and modifying the compiler, as described next in Section 3.2, alleviated the situation.

3.2 Modifications to the Java compiler

The problems encountered were mostly related to the Java front end and not to the rest of the compiler; this is a strong indication of the very high quality of the GCC compiler suite in general. Unfortunately, apparently due to early design decisions, the GCJ compiler is built around the idea that the necessary Java runtime would be mostly written in some other language than Java, e.g., in C++. Considering the fact that the GCJ compiler is able to produce native code in addition to byte code, we considered that approach limiting. Furthermore, as our goal was to implement as much as possible of the RCX operating system in Java itself, we decided to lift these restrictions and modify the compiler so that the runtime could be written in Java to the greatest extent. These modifications are explained next.

Making the compiler-inserted fields visible. In addition to disallowing any new fields from being added to the fundamental classes, the unmodified GCJ rejects (re)definitions of any fields with a name matching any

Field name	Field type and contents
vtable	No changes
monitorSema	byte; id of the thread holding the monitor corresponding to the object
monitorQueue	byte; id of a thread waiting for entry to the monitor; other threads waiting for entry to the monitor are linked to the first thread
monitorCount	byte; the number of times the holding thread has recursively entered the monitor
waitQueue	byte; id of a thread waiting for a notify on this object, any other waiting threads are linked to the first thread

Table 3: Instance fields inserted to the Object class by the modified GCJ

of the compiler generated fields. That is, when we naively tried to add a field named `interfaces` to `java.lang.Class`, the compiler complained about field redefinition. On the other hand, if we tried to use the compiler generated field `interfaces` without explicitly declaring it in the source, Sun `javac` refused to compile the file.

To resolve the dilemma, we modified the compiler so that if the user defines a field with a name clashing with a compiler generated field, and if the types of the compiler generated field and the user defined field are assignment compatible in both directions, the compiler only issues a warning, not an error. By using a new compiler flag, even the warning can be silenced.

Since making the compiler generated fields accessible is only meant to be used in implementing the runtime itself, we went still a little bit further and loosened slightly the assignment compatibility rules. One effect of this was that the compiler still could declare some of the integer fields unsigned (which is *not* representable in Java) while the corresponding user defined fields are signed. Additionally, we allowed the type `java.lang.Void` to function as an unnamed pointer, i.e., something like void pointer in C or C++. Thus, in that way we were able to declare in Java even those compiler generated fields whose types could not be represented as Java types.

In addition to these loosened type compatibility rules, a redefinition in the modified compiler resets the field's visibility to that defined by the user, thereby allowing

Field name	Field type (and contents)
next	a pointer to <code>java.lang.Class</code>
name	a pointer to an UTF8 constant string
accflags	unsigned short, access bits
superclass	a pointer to <code>java.lang.Class</code>
constants	a record enclosing constants info
methods	a pointer to the first element in an array of internal method records
method_count	short, size of the array above
vtable_method_count	short, number of virtual functions
fields	a pointer to the first element in an array of internal field records
size_in_bytes	int, size of instance objects
field_count	short, number of fields
static_field_count	short, number of static fields
vtable	a pointer to the table referred at the Object's vtable
interfaces	a pointer to the first element of an array of class pointers
loader	void pointer, initialized to null
interface_count	short, number of interfaces
state	byte, initialized to zero
thread	void pointer, initialized to null

Table 4: Instance fields silently inserted to the `Class` class

wider access within the runtime. (By default the compiler generated fields are considered private.)

Changing the types of the compiler-inserted fields.

As was explained in Section 3.1, the compiler silently inserts a number of instance fields to the `Object` and `Class` classes, among others. However, the types of many of these fields are defined so that they cannot be expressed in Java. Fortunately, modifying the compiler to use a corresponding Java compatible type was rela-

tively easy in most cases. The new types for the changed inserted fields are given in Table 5. (For the original field types, see Table 4, above.)

Changing the types of some of these fields required considerable changes, some of which were not quite obvious. For example, the old `methods` field was a pointer to a C struct array. Each element in the array described a method, and the field `method_count` provided the length of the array. In our Java friendly version, the `methods` field is a pointer to a Java array object. The generic memory layout of a Java array is described in Figure 2, below. In the standard case, the array object would contain pointers to `Method` objects stored elsewhere in memory. However, since we are here having the compiler generate the array and have full control

Field name	New field type
methods	<code>java.lang.reflect.Method[]</code>
method_count	<i>removed</i>
fields	<code>java.lang.reflect.Field[]</code>
field_count	<i>removed</i>
interfaces	<code>java.lang.Class[]</code>
interface_count	<i>removed</i>
loader	<code>java.lang.Loader</code>
thread	<code>java.lang.Thread</code>

Table 5: Modifications to the fields installed to a `Class`

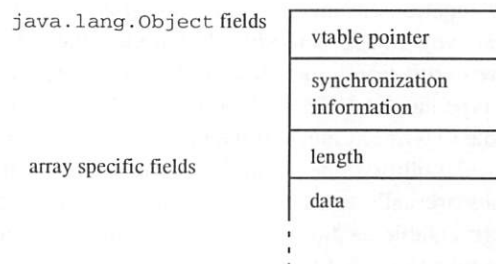


Figure 2: Java array structure

Original function name	The corresponding method after modification	Enclosing class
<code>_Jv_InitClass</code>	<code>void initClass()</code>	Class
<code>_Jv_Register_class</code>	<code>void registerClass()</code>	Class
<code>_Jv_AllocObject</code>	<code>Object allocObject()</code>	Class
<code>_Jv_CheckCast</code>	<code>Object checkCast(Object)</code>	Class
<code>_Jv_IsInstanceOf</code>	<code>boolean isInstace(Object)^a</code>	Class
<code>_Jv_LookupInterfaceMethod</code>	Method <code>lookupInterfaceMethod(String, String)</code>	Class
<code>_Jv_MonitorEnter</code>	<code>static void monitorEnter(Object)</code>	Thread
<code>_Jv_MonitorExit</code>	<code>static void monitorExit(Object)</code>	Thread
<code>_Jv_Throw</code>	<code>static void throwException(Object)</code>	Thread
<code>_Jv_ThrowBadArrayIndex</code>	<code>static void throwBadArrayIndex(int)</code>	Thread
<code>_Jv_exception_info</code>	<code>static Throwable exceptionInfo()</code>	Thread
<code>_Jv_NewArray</code>	<code>static Object newArray(int, int)</code>	Runtime
<code>_Jv_NewObjectArray</code>	<code>static Object newObjectArray(int, Class, Object)</code>	Runtime
<code>_Jv_NewMultiArray</code>	<i>currently unsupported</i>	
<code>_Jv_CheckArrayStore</code>	<code>static void checkArrayStore(Object, Object)</code>	Runtime

Table 6: Runtime functions internally called by the compiler

- a. Note that `isInstance(Object)` is a method that is called not only by the runtime, but that is also available as a part of the public API; in order for this to work, the order of the parameters to the method was changed.

over its internal structure, a more compact representation is possible. Thus, instead of storing pointers to the Method objects in the Java array, we store the Method objects themselves. Since the compiler knows the actual type of the array, it generates correct code when accessed from Java. However, care must be taken when declaring and accessing these kinds of arrays from C++.

Redirecting compiler-generated support functions to Java. Any Java runtime includes a number of relatively high level functions, including memory management, threads, thread synchronization, a few of type management operations, and runtime class handling. To implement these, the GCJ compiler inserts calls to runtime support functions in the generated object code (see Table 6, above). In the unmodified compiler, these functions are C functions, and some of them take arguments whose type cannot be described in Java. Now, in order to be able to write as much of the runtime in Java as possible, we modified the compiler so that the inserted functions are calls to Java methods and the arguments are representable as Java types. The resulting methods are also given in Table 6.

Since these methods are directly inserted to the object code without any access checking, we declared the corresponding methods in the Java classes as using default (package) or `private` visibility, thereby making it impossible to directly use them from outside of the runtime classes. (`Class.isInstance` is an example of deviation from this scheme, as it happens to be a public method defined in J2ME.) In a few cases where the functionality could not be implemented in Java (yet) the corresponding Java methods were declared `native`, and the implementation was made in C++.

Other front end modifications. The other modifications include the following:

- The semantics of `synchronized native` methods and `static transient volatile` variables were changed, as described in Sections 3.3 and 5.4.
- The generation of metadata for Java methods, fields, and class names was made optional. The method and field information is only needed for the reflection API while class names are also needed for the `Class.forName` method. Since most

embedded programs do not use the reflection API nor the `forName` method, leaving that information out reduces the code size about 30%.

- The automatic inclusion of compiler generated fields in the `Object` and `Class` classes was made more generic, or less hardwired, thereby making it easier to specify runtime specific fields in these classes.
- In a number of locations, the front end created code that made computations using the Java `int` type variables. For example, the `array length` field and any computations applied to it, including array index checking, used `ints` internally. However, since the RCX is a 16 bit processor, and has only 64 kB of total address space, the resulting assembly code is both inefficient and unnecessarily voluminous. To alleviate this problem, we created a new compiler symbol for Java array lengths, and used that in the calculations. Selecting an 16 bit integer for this type produced much better assembler code.
- According to the Java specification, the JVM handles integral data types that have less than 32 bits as 32 bit integers during computation of expressions. However, GCJ attempts to optimize this issue and uses only the native machine word size whenever it is sufficient. Now, when GCJ reads in Java byte code, it cannot immediately determine the “real” types of JVM stack variables just by inspecting the byte code instructions. Handling of this had a couple of bugs that appeared only in machines with a small native word size. The fixes were contributed back to the GCJ project. However, more could be done in this respect, and new optimizations would help produce even better code.

3.3 H8/300 back end optimizations

The backend optimizations we made were mostly related to two issues. First, we changed the calling conventions to produce slightly better code and to be compatible with the RCX ROM calls. Second, we changed the register usage directives so that the resulting code uses fewer memory references than with the default directives, when compiling typical Java code. These changes required us to modify the register usage of the runtime system, including `setjmp` and `longjmp` as well as exception handling.

Changes in register usage. The standard GCC back end for the Hitachi H8/300 microcontroller uses register `r7` as the stack pointer, `r6` as a frame pointer, and registers `r0...r2` to pass the first three arguments to functions. The rest of arguments, if any, are passed on the stack. While a standard practice, this has a drawback in

the case of Lego RCX, since the RCX ROM calling conventions are different. That is, the first argument to a ROM routine is passed in register `r6` while the rest are passed on the stack. LegOS and `librcx` have solved this problem by adding a small assembler wrapper which is used when calling the ROM routines. However, in order to gain efficiency and to minimize the amount of code not written in Java, we solved the problem differently.

Basically, GCC calling conventions are defined using relatively simple macros and functions in the target specific back end specification. Thus, in order to support direct ROM calls from Java we created a new back end variation, called `h8300-hitachi-rcx` (instead of default `h8300-hitachi-hms`), that has a different call convention. First, register `r3` is used as a frame pointer instead of register `r6`. Second, the first parameters to a function are passed in registers `r6`, `r5` and `r4`, *unless* the call is declared as a ROM call, in which case only `r6` is used to pass parameters, and the rest of parameters are passed on the stack. As other register usage related optimizations, we declared that frame pointers may be omitted when not needed and that register `r4` is saved over function calls. Together these reduced both code size and the number of memory stores and loads.

Calling ROM functions directly. The changed register usage made it easy to call ROM routines directly from C by declaring a ROM routine as an external function and declaring the actual address in the linker configuration file. To ensure correct parameter passing, the external function declaration must be announced as a ROM call by using a C or C++ `__attribute__`, or a corresponding preprocessor `#pragma`, which signals the back end to use the alternative calling conventions.

As an example, let us consider the ROM routine “set LCD segment,” which is available at `0x1b62`. The corresponding C declaration and a relevant fragment of the linker configuration file are given in Figure 3, on the next page.

Making a ROM routine callable directly from Java required a little bit more. First, Java does not include any feature corresponding to the C/C++ usage of attributes in method or field declarations. However, Java allows a native method to be declared synchronized, but does not associate any specific semantics for this. (Synchronization is implemented by a method itself; the caller of a method does not need such information.) Thus, we hacked the compiler so that it flags any method declared as native synchronized as a ROM entry point.

```
extern char set_lcd_segment(
    short code
) __attribute__((ROM));
```

```
SECTIONS {
    ...
    .rom : {
        _set_lcd_segment = 0x1b62;
        ...
    } > rom
    ...
}
```

Figure 3: A C declaration for an RCX ROM routine, and a corresponding linker directive

Second, the link time naming conventions for Java methods are different from those of C functions. In order to overcome this, the entry points must be declared as *mangled names* in the linker configuration file. The result of converting the C example is shown in Figure 4, below.

4 Minimal Java runtime

Originally, the Java 1.0 and Java 1.1 specifications defined a single language and runtime structure. However, along with the success of Java, Sun Microsystems started to define a number of versions of the runtime, targeted for different purposes. Currently, *Java 2 Micro Edition* (J2ME) [15] is the smallest runtime specification still supporting the full language. On the other hand, the *Java Card Runtime Environment* (JCRE) [16] provides a still smaller runtime, but due to the restrictions it imposes on the language it is arguable whether JCRE utilizes Java at all in the sense that the rest of the Java runtime environments do.

```
class Display {
    static native synchronized void
        setSegment(short code);
}

SECTIONS {
    ...
    .rom : {
        _setSegment__Q17Displays = 0x1b62;
        ...
    } > rom
    ...
}
```

Figure 4: A Java declaration for an RCX ROM routine, and a corresponding linker directive

In our project, the goal is to be as compatible as possible with J2ME. However, the J2ME specification is based on the assumption that the runtime environment is capable of executing Java class files, i.e., has an interpreter, JIT, or hardware support for byte code. Due to space restrictions, this is not possible in the RCX, and therefore we cannot be fully compatible with J2ME.

Below, we describe the basic API provided by our environment, the language features currently not supported, static constructor, destructor and class initialization routines, and thread support. The description of the operating system level features are deferred to Section 5.

4.1 java.lang APIs

The `java.lang` classes included in our runtime are enumerated in Table 7. The methods provided are more restricted than they are in the Java standard edition, but mostly aimed to be compatible with J2ME. However, Sun has not produced any public specification of the actual J2ME API. Therefore, we have used Embedded Java [17], which is available, as our comparison point. The most important differences from the Embedded Java API are outlined in the table. Most notably, the `String` class is heavily restricted, providing only minimal support. Furthermore, some exceptions and many errors associated with runtime checks are eliminated due to the static nature of our runtime environment.

When compiled into class files, the total size of the class files is about 32 kilobytes. As a binary library (including symbols), the runtime fits into 170 kilobytes. In binary, with all the symbolic and metadata information eliminated, a typical runtime size is 15 kilobytes.

4.2 Language level restrictions

Our current implementation has a number of restrictions that affect the typical programmer. These include the following.

- The runtime library does not currently support floats or doubles. Source code attempting to use them compiles but does not link.
- Multidimensional arrays are unsupported. Code attempting to create such arrays calls `newMultiArray`, which has not been implemented.
- Java interface classes are not supported either, since their current implementation in GCJ relies on the method and field metadata, and we want to omit them from the binaries. If the metadata information is included, the interface support should work, but that has not been tested.

Classes	Differences to Embedded Java
Object	Method toString omitted; otherwise full API.
Class	Methods getResource, getResources, and getSigners omitted; otherwise full API.
ClassLoader	Only available as a placeholder. All methods omitted.
System, Runtime	I/O channel, library, process, property, security manager, and trace related fields and methods omitted. Some of the other methods are currently placeholders only.
Throwable, Error, Exception	No messages or stack trace supported. All errors and exceptions are assumed to be immutable singletons. Each class has a constant static field instance.
Void	Full API. Also used as an opaque type for data whose type is not available in Java.
Boolean, Byte, Integer, Number, Long, Short	String related parsing and printing omitted. Conversions to floating point numbers currently unsupported. Otherwise full API.
Double, Float, Math	Currently unsupported.
Character	Only rudimentary support. Most methods omitted.
String	Only rudimentary support. No constructors nor any methods creating new strings are available. All strings must be created at the compile time.
Thread	String, security manager, stack trace, thread group, and daemon threading methods omitted. Priorities are currently unsupported, but the API is available.
Compiler, Process, StringBuffer, Security-Manager, ThreadGroup	Not available at all. Thread groups might be added later; no need for others.

Table 7: Differences between Embedded Java API and the API of our implementation
(Some classes, such as the exception and error classes, are left out for brevity.)

4.3 Static constructors and destructors

When compiling Java code, the GCJ compiler produces stubs for class registration. That is, for each compiled class, the GCJ produces a piece of code that is not called anywhere from the code, but a pointer to it is placed into a so called .ctors loader area. The initialization system in the runtime loops over the pointers in the .ctors area, calling each function in turn. The same mechanism is used for running static constructors in a C++ program. In the case of GCJ generated code, however, the stub code only calls the registerClass routine, allowing us to create a list of all classes.

4.4 Class initialization

By default, the GCJ compiler behaves strictly according to the JVM specification, and generates lots and lots of calls to initClass. That is, whenever a Java class is accessed from the Java source code, the code generator generates a piece of code that first calls initClass, and then performs the actual class access. For example, consider the following code fragment.

```
class Ex1 { static short x; }
...
    Ex1.x = 0;
...
```

The code generated from the assignment looks like the following.

```
mov.w    r6, #_CL_Q13Ex1
jsr      initClass
sub.w    r0, r0
mov.w    r0, __Q13Ex1$x
```

This approach complicates slightly the writing of the code for the initClass method. That is, care must be taken that all classes that are accessed from initClass, either directly or indirectly, must be separately initialized by explicitly marking them initialized and calling their class initializers *before* any other Java code is called. Furthermore, the class initializers of these classes must not cause invoking of the initClass method for any other classes. If these rules are not strictly followed, a call to initClass easily results in an unterminated recursion or other disastrous complications.

4.5 Memory management

As already mentioned, most of the runtime environment and operating system was written in Java. However, explicit C, C++ and even assembler support was needed for the memory management, exception handling, thread management, and thread synchronization. Of these, exceptions, thread management and thread synchronization are discussed in more detail later in Section 5, while memory management is considered next.

In Java, the memory is managed at the granularity of objects. Each object consists of a few compiler generated fields (see Table 3) along with any user defined instance variables. Each instance variable contains either a reference to some object, or a primary data item. For the variable fields, GCJ uses natural field sizes, and aligns the fields according to the C++ alignment rules (this is different from what most JVMs do). In order to allocate objects, the compiler arranges calls to allocation routines `allocObject` and various versions of `newArray`.

Due to the limited amount of memory available and the nature of programs running in a typical robotics application, we assume that most applications will create a small number of relatively long living objects. Furthermore, temporary circular data structures will be more likely the exception than the rule.

Our current garbage collection method is very simple, and similar to the Java Card Runtime Environment. That is, garbage is not collected, and allocated objects stay around until the next firmware or hardware reset. However, to better support dynamic data structures we are planning to support garbage collection. The current options include a simple reference counting scheme that would be implementable in the compiler back end, and a simple mark-and-sweep collector utilizing object layout information provided by the compiler. However, both of these schemes require considerable support from the compiler, and are presently left for further study.

5 Operating system services

Due to the simple nature of the RCX hardware, the border between the Java runtime environment and the operating system is ambiguous. However, while object level memory management along with garbage collection is more or less hardware independent, exceptions, threads,

low level hardware access, and events are more tightly bound to the underlying hardware than to the language. Therefore, we decided to classify the latter issues as operating system services, and consider them next.

5.1 Exceptions

The GCC collection of compilers have two different possibilities for exception handling. The default method uses the `setjmp` and `longjmp` functions while the alternative method explicitly scans the execution stack looking for exception handling frames. Since the default method produces more compact code, especially when we told the compiler to use our own versions of `setjmp` and `longjmp` instead of the compiler internal versions, we decided to use the default method.

In Java, there are two basic constructs that cause the compiler to construct exception handling frames. The first one is obvious, namely the Java `try ... catch` construct. The other case are the `synchronized` methods and statements, which use exception handling code to release monitor locks.

Now, whenever a GCC compiled function establishes an exception handling frame, it first calls an internal function that returns an *exception context*. In our implementation, the context is a part of the currently running thread. Next, when entering the protected block of code, the compiler allocates a few bytes on the stack, calls `setjmp` to store the current execution context there, and links this stack frame to the front of a list of exception handling frames, available in the exception context.

If the execution of the block terminates normally, the stack frame is popped from the list and the stack is restored. On the other hand, whenever an exception is thrown, the exception context is consulted to get the list of exception handling frames, and each frame is called in order until the exception is caught or the list terminates. In the latter case, the execution of the thread is terminated.

5.2 Threads and synchronization

In the RCX, a thread of control is very simple, essentially consisting of the current processor state. A context switch merely changes the contents of the processor registers. The state is stored in a `Thread` instance, and is visible to the Java environment as a series of volatile short instance variables. This allows direct manipulation of non-running threads from Java code.

To save memory and simplify implementation, the number of threads is limited to 126. This allows a thread identifier to be stored in a single signed byte (thread number zero is used as a sentinel). Each thread also contains a link byte, possibly containing a thread ID of another thread. These link bytes are used to create lists of threads waiting for a specific event.

Monitors. Object level synchronization, implemented in `Thread.monitorEnter` and `Thread.monitorExit`, is implemented with a simple per object semaphore together with a counter and a queue. The `monitorEnter` and `monitorExit` actions disable interrupts by manipulating the processor condition code register through the `Thread.disableInterrupts` and `Thread.enableInterrupts` assembly routines.

The monitor semaphore is represented as a single byte, present in all objects (see Table 3). Whenever there are no threads within the object's monitor, the semaphore byte is zero. When the first thread enters the monitor, it places its thread ID to the semaphore byte. Any other threads attempting to enter the monitor will find the monitor occupied, and insert themselves to the head of the monitor queue.

When a thread leaves the monitor, the thread checks if there are any other threads waiting for the monitor. If such a thread is found, the semaphore is kept busy and the first waiter is removed from the queue and its ID is placed into the semaphore byte, after which it is woken up by linking it to the run queue. On the other hand, if there are no waiters, the semaphore is simply released.

Since a Java thread can recursively enter a monitor several times, yet another variable is needed to keep count of these recursive entries. Due to the limited stack in the RCX, we decided to use a single byte as this counter as well.

Condition variables. Another type of synchronization is provided by the Java `wait` and `notify` primitives. In Java, any thread holding an object monitor may invoke the `Object.wait` method. This suspends the thread issuing the call until either another thread invokes the `Object.notify` method for the same object, or a time out occurs.

Again, our implementation is fairly simple. In addition to the monitor thread queue, each object also includes a waiter queue. This queue is also implemented as a simple byte, which is initialized to zero. Whenever a thread enters a `wait`, it stores the current value in the wait

queue to its thread ID link, and places its own thread ID in the wait queue byte; this, effectively, places the thread at the head of the waiter list.

When another thread invokes `notifyAll`, thereby waking all threads waiting on a thread's wait queue, the thread list is simply moved to the monitor queue. The `notify` method, instead, just moves the head of the wait queue to the monitor queue. (In this case the wait queue acts as a stack, but that is perfectly fine according to the Java language specification.) The notified thread or threads are woken when the notifying thread leaves the monitor, as explained before.

Wait time-outs are currently not supported; their addition may require slight revisions to the implementation.

5.3 ROM services

The ROM services are encapsulated into a number of platform specific classes, enumerated in Table 8. IR communication is not supported, yet. The platform specific classes form a package of their own, called `com.rcx`.

Class	Description
Button	Initialize, read and shutdown RCX buttons.
Display	Modify the LCD screen contents.
Motor	Power motors in a controllable way.
Power	Allows access to power savings.
Port	Direct access to the hardware I/O registers.
Sensor	Power, read, and shutdown sensors.
Sound	Allows sounds to be played.
Timer	Access to the hardware timers.
Vector	Direct access to the interrupt vectors.

Table 8: Platform specific classes in the `com.rcx` package.

Most of the platform specific classes contain a number of native synchronized methods that are used to directly call the ROM routines. In most cases, these methods are `public`, allowing direct access from anywhere in the program. Only those ROM calls that require specific arguments or otherwise cannot be called without possible problems are protected by restricted visibility.

5.4 Hardware access

In some cases it is clearly beneficial to bypass the ROM and to access the hardware directly. To support this, we modified the GCJ compiler so that any variable defined

```

class Port {
    ...
    static transient volatile PORT4;

SECTIONS {
    ...
    .eight (0xFF00) : {
        _Q14Port$PORT4 = 0xB7;
    } > eight
    ...
}

```

Figure 5: Definition of the variable `Port.PORT4`, which allows direct access to the hardware I/O register number 4.

as `static transient volatile` is considered as if it were a declaration of an external variable instead of being a definition. Thus, the compiler does not allocate any memory for those kinds of variables. Therefore, their location in the memory can be freely decided by the linker. Again, utilizing linker directives made this easy. Figure 5 illustrates the definition of I/O Port 4, whose memory address `0xFFB7`. Among other things, port 4 can be used to directly read the status of two of the user buttons.

5.5 Event model

In Java, it is customary to represent changes in external environment as events. Thus, our intention is to abstract changes in button and sensor values as events. The basic idea is to have a thread polling on a specific I/O port, waking up on interrupts generated either by a change in the port or by a timer. If the polling thread notices that the value represented at the port has changed, it takes an event object from a queue of free event objects, fills in the appropriate values, and places the object in an appropriate event queue. A non-interrupt level thread would be waiting on the queue, and handle the event after the interrupt routine has been completed. Finally, the event would be passed back to the free event queue when it is not needed any more.

5.6 Other services

We expect to enhance our operating system with a number of additional services. The planned services include communications (both basic IR and IP over IR) and scheduled power management (to save power in long running sensor-type applications). However, these features are currently left for further study.

6 Evaluation and lessons learned

It was no surprise to us that it was both challenging and fun to write a new operating system (or an operating systlet) in a new language for a hardware we were not familiar with when we started. However, the main obstacles came from a direction we could not anticipate. That is, the intrinsic interrelationships between a compiler and the corresponding runtime system are much more complex and fragile than we originally expected. Writing a runtime system independent compiler for C is clearly feasible, as shown by GCC. The same applies, more or less, to the GCC C++ compiler. However, the current GCJ compiler is far from being runtime neutral, and currently one is required to have good knowledge of the compiler internals in order to be able to write a new type of runtime system. We learned this the hard way, and hope that this paper and our modifications to the GCJ allow others to do the same more easily.

Looking at the situation from another direction, the fact that we were able to complete the project in the first place is an indication of the feasibility of our approach. First, we have shown that the idea of using Java as a low level language to implement both a Java runtime environment and a minimal operating system in Java is viable. The basic methods, or tricks, that we used in our implementation include the following:

- Using a compiler that produces native code instead of byte code.
- Wiring the compiler-generated Java runtime primitives (see Table 6) back to Java, thereby allowing the primitives to be implemented in Java instead of some other language.
- Enabling Java source level access to the meta-information generated by the compiler. This allows, for example, an easy pure Java implementation of `Class.isAssignableFrom`.
- Converting `static transient volatile` and `synchronized native` modifiers into external declarations and back end specific attributes, respectively, which make it possible to tailor the compiler and linker to provide direct access to the underlying ROM routines and hardware addresses.

Second, our experience indicates that writing an operating system in a beautiful object oriented language, such as Java, gives a number of benefits. In the present case, the target environment is such a simple device that a strict boundary between the operating system and an application would probably only complicate things and make the application both bigger and less efficient. Object-orientation allows some of the underlying problems to be solved in a neat way. That is, visibility rules, data

hiding, and inheritance make it possible to provide an application programmer an environment where the application may be tightly integrated with the operating system without compromising architectural layering or introducing unnecessary bugs. We allege that the same principles could also be applied in a more complex case if appropriate memory management hardware was added.

Considering the language, the main benefits of Java lie in its relative strictness when compared with C++. That is, given any C++ based operating system level framework, the programmer is required to understand considerable amount of the implementation details in order not to mistakenly break the underlying semantic assumptions of the framework. In the case of Java, the semantics-related problems are easier due to the more strictly defined language specification and fewer possibilities for a programmer to circumvent language-level object abstractions.

The use of Java brings up another benefit. That is, since the APIs are to a large extent compatible with the standard Java APIs, it should be possible to port a large number of Java packages to the RCX with no or minimal changes. Now, for example, porting a minimal JACL [18] interpreter to the RCX should not be too hard.

Hence, our work has shown that Java can be used as a viable language for low level programming, with benefits unavailable from other approaches.

7 Future work

At the present time (April 2000), garbage collection, threads and the event model require more work. Some of the modifications made to the GCJ compiler could be made more generic and supplied back to the standard version of GCJ. An extension to study is the ability to handle dynamically loaded code based on the work recently introduced in LegOS. However, due to the Java visibility constraints this may not be easily adoptable.

Once the basic operating system platform has stabilized, we plan to focus on communication issues. The aim is to port our Java Conduits Beans (JaCoB) protocol framework [19] to the RCX, and to build a minimal IPv6/UDP implementation on the top of that. Our hope is to see if it would be possible to make the RCX robots first class citizens in Jini communities.

Availability

The source code for the system is available at <http://www.tcm.hut.fi/~pnr/rcx/>. The actual source tree is supplied as a gzipped tar file, whose size is about 250 kilobytes. Building the system requires patched versions of both GNU binutils and GCC; the necessary patches are provided. The binutils patch is minimal (less than 2 kilobytes) and should not cause any problems. However, since the various versions of the GCC patch are fairly large (about 150 kilobytes) and since they were made against GCC-current instead of any specific released version, building a working compiler may require some manual work, or, alternatively, using CVS to check out GCC-current of the date when the particular version of the GCC patch was created.

Acknowledgments

This work would have not been possible without the large number of people working on the RCX reverse-engineering and the various programming environments, including, in no particular order, Kekoa Proudfoot, Markus L. Noga, David Baum, Peter Liu, Stephen Spackman, Michael Daumling, Ross Paterson, Frank Cremer, Sergey Ivanyuk, Mark Falco, Mario Ferrari, Frank Mueller, Tom Emerso, Lou Sortman, Luis Villa, David Van Wagner, Michael Nielsen, Chris Dearman, Eric Habnerfeller, and Ben Laurie.

Since the availability of the compiler source code was essential for this work, we want also to thank Richard Stallman, the Free Software Foundation, Cygnus Solutions (now part of Red Hat), the GCJ implementation team including Alexandre Petit-Bianco, Per Bothner, Andrew Haley, Tom Tromey, Anthony Green, Warren Levy, Bryce McKinlay, and others, and the large number of volunteers for their work in providing free software in general, and the GNU Compiler Collection in particular.

We are also grateful to Tuomas Aura, Hannu Napari and Lauri Savioja of HUT and Chris Demetriou and the anonymous reviewers of USENIX for their comments and suggestions how to improve the paper, to our students Markus Aholainen and Veera Lehtonen for their feedback about some early versions of the system, and especially to Petri Aukia of Bell Labs for his numerous constructive suggestions concerning this work in its early stages.

References

- [1] *Lego Mindstorms*,
<http://www.legomindstorms.com/>
- [2] *The MIT Programmable Brick*,
<http://el.www.media.mit.edu/projects/programmable-brick/>
- [3] Kekoa Proudfoot, *RCX Opcode Reference*,
<http://graphics.stanford.edu/~kekoa/rcx/opcodes.html>
- [4] Lego RCX Code, in *Robotics Invention System User Guide*, The Lego Group, 1998.
- [5] *Lego Dacta RoboLab*, <http://www.lego.com/dacta/roboLab/default.htm>
- [6] David Baum, *Not Quite C (NQC)*,
<http://www.enteract.com/~dbaum/nqc/index.html>
- [7] Markus L. Noga, *LegOS Home Page*,
<http://www.noga.de/legOS/>
- [8] Kekoa Proudfoot, *Librcx*,
<http://graphics.stanford.edu/~kekoa/rcx/tools.html#Librcx>
- [9] *GNU Compiler Collection (GCC) home page*,
<http://gcc.gnu.org/>
- [10] Michael Durrant and D. Jeff Dionne, *uCsim Home Page*, <http://www.uclinux.com/uC68EZ328/index.html>
- [11] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath, *The Jini™ Specification*, Addison-Wesley, Reading, MA, July 1999.
- [12] Kekoa Proudfoot, *RCX Internals*,
<http://graphics.stanford.edu/~kekoa/rcx/index.html>
- [13] *Hitachi Single-Chip Microcomputer H8/3297 Series*, <http://semiconductor.hitachi.com/products/pdf/h33th014d2.pdf>
- [14] Gintaras R. Gircys, *Understanding and Using COFF*, O'Reilly & Associates Nutshell Series, Sebastopol, CA, 1988.
- [15] *Java™ 2 Platform, Micro Edition (J2ME)*,
<http://java.sun.com/j2me/>
- [16] *Java Card™ Technology*,
<http://java.sun.com/products/javacard/>
- [17] *EmbeddedJava™ Technology, Source Edition*,
<http://www.sun.com/software/embeddedjava/>
- [18] Ray Johnson, *Tcl and Java Integration*, Sun Microsystems Laboratories, Palo Alto, CA, Feb 1998.
- [19] Pekka Nikander and Juha Pärssinen, "A Java Beans Framework for Cryptographic Protocols," in Mohammed Fayad, Douglas Schmidt and Ralph Johnson (Editors), *Object Oriented Frameworks, Volume II*, Wiley, 1999.

LAP: a little language for OS emulation

Donn M. Seeley

Berkeley Software Design, Inc.

ABSTRACT

LAP, the *Linux Application Platform*, is a Linux emulation package for BSD/OS which uses a “little language” [Salu98] to describe transformations from Linux data types and values to BSD/OS data types and values, and *vice versa*. The little language simplifies and regularizes the specification of transformations, making the emulation easier to maintain. This paper describes the language and its place in the framework of LAP.

1. Introduction

The **Linux Application Platform** is a package of software that allows Linux [Linu00] applications to run under BSD/OS [BSDI00]. Although BSD/OS and Linux share a common executable file format (the Unix **ELF** format [Unix90]) and a common Unix-like programming interface (based on the IEEE **POSIX** interface [IEEE96]), they differ in the way that applications communicate to the operating system kernel, and they differ in the sizes and structure of their data types. The LAP software dynamically converts Linux data types into BSD/OS data types and *vice versa*, and it substitutes a BSD/OS kernel interface for the application's Linux kernel interface.

The LAP software uses *transformations* to convert data types, system call numbers and other parameters between Linux and BSD/OS. Transformations are written in a *transformation language*. The language looks a lot like C [ANSI89]; the inspiration for its syntax comes from **lex** [Lesk75]. The language specification tries to make the most common transformations also be the simplest ones to specify. Many transformations can be described by a simple C prototype declaration.

In the sections below, we'll discuss some of the design issues that we considered when writing the LAP software, and we'll introduce some of the features and describe how they work.

2. Motivation

2.1. Why do we need transformations?

Here are a few examples of the differences between BSD/OS and Linux which require transformation:

- BSD/OS follows the Unix Application Binary Interface standard [Unix90] for system calls, using the `lcall` instruction to transfer control to the operating system kernel. Linux uses the `int` instruction to transfer to the kernel, which generates a software interrupt, more like MS-DOS.
- The operating system assigns numbers to system calls, signals and error conditions, and BSD/OS and Linux use different numbers.
- The POSIX application programming interface requires certain abstract data types, which are defined using the C `typedef` mechanism. The BSD/OS and Linux systems use different real types for these data types for some interfaces. For example, the `uid_t` type describes user ID values; in BSD/OS, these values are unsigned 32-bit integers, while in Linux these are unsigned 16-bit integers.
- The POSIX API also requires certain aggregate data types which correspond to `struct` data types in C. The order of elements or the types of elements in a `struct` may be different in BSD/OS and Linux. For example, the file status structure `struct stat` in BSD/OS puts the timestamps before the sizes, while in Linux the sizes come first.

2.2. Why use a transformation language?

The most important reasons for using a transformation language are reliability and maintainability.

The transformation language makes transformations *reliable* by removing opportunities to commit errors in specifications. BSD/OS also has an emulation for SCO Unix SVr3.2; it uses a very primitive transformation table that required a lot of hand-composed support code. While the SCO emulation's transformations were reasonably well structured, using strict conventions for naming, parameter types and ordering, errors crept in, and the errors were sometimes very difficult to diagnose. A transformation compiler makes a lot of the “mechanical” work truly mechanical, reducing a class of errors.

The transformation language makes transformations *maintainable* by making them familiar and easy to read. The language looks like C and (for the most part) works like C; it's straightforward to read a transformation, and it's usually obvious what it does. When transformations are composed by hand, they are more difficult to

read and understand, and hence the code is harder to fix. Of course the transformation compiler itself can have bugs, but its bugs usually affect groups of system calls rather than individual system calls, so they are easier to spot.

Other applications have used specialized languages for similar reasons. The original idea for the LAP transformation language comes from code template languages for compilers. The first such code template language that I used appeared in the Ritchie C compiler for 6th Edition Unix [Ritc79]; the Portable C Compiler [John79] and the GNU C Compiler [Free00] also have code template languages. One very direct inspiration was the master system call table in BSD-derived operating systems (`/sys/kern/syscalls.master`).

2.3. Design issues

The principal design goals for LAP were to:

- keep the specification simple
- keep the implementation simple
- make few modifications to Linux code
- maintain efficiency
- avoid burdening the operating system kernel

Keep the specification simple: If we can make the specifications for most transformations very simple, then we are likely to commit fewer errors when writing transformations. Also, the effort of writing transformations is reduced.

Keep the implementation simple: The code uses a **yacc** parser [John75] and a **lex** scanner to implement the transformation compiler. The output of the transformation compiler is more-or-less readable C, which we then compile using GCC. We use Berkeley DB [Olso99] to hold symbol databases. By using tools and keeping the code simple, we leave fewer opportunities for mistakes and spend less time on maintenance.

Make few modifications to Linux code: By interposing a relatively small number of low-level interfaces in front of actual Linux shared libraries, we reduce the chances that we will interfere with interactions between the application and the libraries, and we reduce the amount of work that we set for ourselves.

Maintain efficiency: We try to avoid design decisions that would cause us to add overhead by requiring us to block signals to protect internal data structures or make other expensive accommodations.

Avoid burdening the operating system kernel: We want to avoid changes to the BSD/OS kernel to support Linux emulation. This means that extra kernel resources don't need to be tied down for Linux emulation. It also means that installing new LAP software does not require an update to the kernel. It means that

debugging is simpler, and that it's unlikely that an error in LAP software will cause the operating system to crash.

3. The LAP software

Before going into details of the transformation language, let's briefly look at the big picture - what is LAP and how does it work?

LAP changes the execution environment for Linux applications so that they can satisfy their requests for services using the native BSD/OS kernel, rather than a Linux kernel. LAP's operation is based on *dynamic linking* [Ging89]. When a Linux application runs, the operating system loads it into virtual memory along with a separate program known as the dynamic linker or **ld.so**. The dynamic linker is responsible for loading *shared libraries*, which are collections of useful executable code that the application needs in order to communicate to the operating system (among other things). LAP replaces the Linux dynamic linker with a lightly modified version that can make native BSD/OS system calls, and interposes a library named **liblinux** in front of unmodified Linux libraries. The **liblinux** library overrides the Linux libraries and causes the application to make calls into the native BSD/OS kernel rather than into a Linux kernel.

Both the modified **ld.so** and **liblinux** are built from source code written in C, assembly language and the transformation language. A separate *transformation compiler* converts the transformation language into C, which is in turn compiled by the native C compiler. LAP does not create or interpret transformations at run-time. Apart from its own libraries, LAP does not use any native BSD/OS libraries - all other shared libraries in the LAP environment are unmodified Linux libraries.

4. The transformation language

The transformation language resembles C, with inspiration from **lex**. Source code in the transformation language is translated into C by the **transform** program. By convention, source code files for transformations end in the suffix `.x`, while headers for transformation sources end in `.xh`. Transformations describe how to convert between Linux data types and system calls, and BSD/OS data types and system calls.

As an example, here is a transformation for the `stat()` system call:

```
int stat(const char *name, struct stat *buf);
```

Given the appropriate `stat.xh` header, this transformation causes **liblinux** to do the following:

- Execute an `lcall` instruction to perform the BSD/OS `stat()` service, placing the resulting data in a buffer on the stack.
- Convert the BSD/OS `stat` structure into a Linux `stat` structure in memory belonging to the application.
- If there is an error, convert the BSD/OS error code into a Linux error code and store it in the Linux `errno` location.

The following sections provide a more detailed description of the transformation language.

4.1. Lexical structure

The basic elements of the transformation language are similar to C. Unlike C, there is no macro preprocessor; however, it is possible to “escape” to C code and write C preprocessor code in that context.

The language defines *keywords* that introduce statements or qualify declarations. All C keywords are reserved. Several keywords that are specific to the transformation language are introduced in the syntax section below, along with the statements that use them. The `typedef` statement provides a mechanism for defining new keywords, analogous to the C `typedef` statement.

There are *names* and *numbers* that work much like they do in C. Names are introduced in C-like contexts such as function names, parameter names, structure tags, structure members, and so on. Names follow the usual C rules - they must begin with a letter or an underscore, and may contain letters, digits or underscores. Numbers also follow C rules; the **transform** program simply passes numbers into its C output without interpretation.

The transformation language treats specially those names that begin with a *foreign* or *native* prefix. Names that start with `LINUX_`, `linux_` or `__bsdi_` cause the **transform** program to place restrictions on the automatic mapping between Linux names and BSD/OS names. See below for more information on this feature. The transformation language also recognizes the special prefix `__kernel_` on function names; more on that below as well.

Strings in the transformation language are used only to give the names of header files. They are surrounded by double quotes and they don’t follow C rules for escapes (yes, very crude).

Various *punctuation marks* and “syntactic sugar” are recognized, including parentheses, commas, semicolons and braces. Certain punctuation implies an escape to C. As in **lex**, text that appears inside percent-brace pairs `%{ ... %}` is treated as literal C, and text that appears between simple braces following a function

declaration is also treated as C. This C text is included in the output from the **transform** program without significant alteration.

Comments and *whitespace* are basically the same as C: text inside slash-star and star-slash `/* ... */` is ignored, and comments, spaces, tabs and newlines serve to break input into tokens, but are otherwise collapsed together and ignored.

4.2. Syntax

The syntax of the transformation language is organized into *statements*. The transformation language itself provides only declarative statements. Any imperative statements must be coded in C inside C escapes. Here is a summary of the statements. Literal text appears in *fixed width font*, while text that varies appears in *slanted fixed width font*.

Include

```
include "header"
```

The `include` statement causes text from the named header file to be inserted into the program text at the current location. Note that there is no “#” character at the beginning of the line. If a program needs to include a C header so that text in C escapes can use the header information, then that C header must be included using a C escape too; for example, `include` includes a transformation language header, while `%{ #include` a C header.

Typedef

```
typedef type-specifiers ... name;
typedef type-specifiers ... name {
    in(name) { ... }
    out(name) { ... }
};
```

The first form of the `typedef` statement looks much like a C `typedef`. It declares *name* as a type name. If *name* begins with `linux_`, then the type is a *foreign* type with no BSD/OS equivalent; otherwise, the **transform** program creates a mapping between the given type name in Linux and the type with the same name in BSD/OS. In the latter case, there really are two types, but the difference is hidden by the mapping feature. Inside C escapes, the BSD/OS type has the usual name while the Linux version of the type is prefixed with `linux_`. Note that the transformation language does not define the BSD/OS version of a type name; you must provide that yourself in a C escape, either by including the appropriate header file or by writing an explicit C `typedef` statement.

As an example, the statement `typedef unsigned short uid_t;` in the transformation language says that there is a Linux type named `uid_t` that corresponds to a BSD/OS type `uid_t`, and that it is equivalent to the basic C type `unsigned short` in Linux. When `uid_t` is used in a parameter list or a structure definition, the Linux value is automatically copied (as if by assignment) into the corresponding BSD/OS value on input (which has a type corresponding to `unsigned int`), and the BSD/OS value is automatically converted into the Linux value on output. For example, the transformation `int setuid(uid_t uid);` converts the Linux `uid` value into a BSD/OS `uid` value before calling the BSD/OS system call.

The second form of the `typedef` statement allows you to specify *transformation functions* for the given integral type. The `in()` function is automatically called to convert Linux types into BSD/OS types, while the `out()` function is automatically called to convert BSD/OS types into Linux types. The parameter *name* represents the value to be transformed. The body of the function is given in C inside braces. One or both transformation functions may be omitted, in which case the value is transformed by assignment. As an example, the statement

```
typedef unsigned short dev_t {
    in(dev) {
        return (makedev(dev >> 8, dev & 0xff);
    }
};
```

specifies an input transformation for `dev_t` that converts Linux `dev_t` values into BSD/OS `dev_t` values using the BSD/OS `makedev()` macro. Note that a `typedef`'s transformation functions may be accessed directly inside C escapes by appending `_in()` or `_out()` to the type name; this is true of transformation functions in general.

Cookie

```
cookie type-specifiers ... name {
    name number;
    in(name) { ... }
    out(name) { ... }
};
```

The `cookie` statement is an enumeration statement that creates a type like a `typedef` and lists members of that type along with their Linux values. When an object of the given integral type appears in an input context and its value matches one of the enumerated values, that value is converted to the value with the corresponding name in BSD/OS. This is a fancy way of saying that cookies convert `#define` macros from Linux values to BSD/OS values and back. Inside C escapes, the Linux member names are prefixed with

`LINUX_`. If a cookie member's name is given with a `LINUX_` prefix, the **transform** program assumes that there is no equivalent BSD/OS value; if the cookie type name itself is prefixed with `linux_`, **transform** assumes that none of the members have corresponding BSD/OS names (and it omits the `LINUX_` prefixes in C escapes). If a value of a given cookie type fails to match any of the listed numbers, the value is assigned without conversion - that means that you don't have to list names that have the same value in both Linux and BSD/OS. However, if there is an `in()` or `out()` function, it applies to unmatched values. This lets you take care of values that have no exact equivalent in Linux or BSD/OS. Note that you are responsible for supplying the BSD/OS cookie member definitions, usually by including the appropriate C header file inside a C escape.

As an example,

```
cookie int reboot_t {
    RB_AUTOBOOT      0x01234567;
    RB_HALT           0xcdef0123;
    LINUX_RB_ENABLE_CAD 0x89abcdef;
};
```

says (among other things) that `RB_AUTOBOOT` has the value `0x01234567` in Linux and that there is no direct BSD/OS equivalent for the Linux name `RB_ENABLE_CAD`. An object of type `reboot_t`, presumably the argument to `reboot()`, with value `0x01234567` would be converted to the BSD/OS value of `RB_AUTOBOOT`, which happens to be 0. (Yes, Linux uses enumerated values rather than flags as arguments to `reboot()`.)

Flag

```
flag type-specifiers ... name {
    name number;
    name;
    in(foreign, native) { ... }
    out(native, foreign) { ... }
};
```

A flag works very much like a cookie but for flag bits rather than enumerated values. Flag values are tested for matches by logically and-ing against the appropriate Linux (on input) or BSD/OS (on output) value. If a match occurs, the corresponding BSD/OS (on input) or Linux (on output) value is logically or-ed in. Bits that aren't matched are copied unchanged, so you don't need to list flag values that are identical on both Linux and BSD/OS. A given input can match more than one flag value. If you provide a transformation function, it gets both the raw value and the converted value, so that you can use a complicated rule to add (or subtract) bits from the converted value after all of the specific conversions are made. If you specify a flag

name without a value, **transform** assumes that the name is a BSD/OS name with no equivalent Linux value. If a member name has a `LINUX_` prefix, **transform** assumes that the name is a Linux name with no equivalent BSD/OS value. Bits that have no equivalent are not copied by default; this is a handy way to clear bits that aren't supported and don't significantly affect the semantics. Inside C escapes, the Linux flag member names are prefixed with `LINUX_`.

Here's an example:

```
flag unsigned int cflag_t {
    LINUX_CSIZE 0000060;
    HUPCL 0002000;
    CRTS_IFLOW;
    in(f, n) {
        return (n | (f & LINUX_CSIZE) << 4);
    }
    out(n, f) {
        return (f | (n & CSIZE) >> 4);
    }
};
```

This flag encodes the flag bits for the `c_cflag` field of a `termios` structure. It says that the HUPCL bit under Linux has the value 02000 rather than 0x4000 as it does under BSD/OS. The BSD/OS CRTS_IFLOW bit has no equivalent under Linux, and we clear it by default in any conversion. The `LINUX_CSIZE` field is also cleared by default, but the transformation functions copy it to and from the BSD/OS `CSIZE` field, so the information isn't lost. Notice how the transformation functions must be careful to preserve the bits that were already converted when returning a value.

Struct

```
struct name {
    type-specifiers ... name;
    type-specifiers ... name[number];
    in(foreign, native, length) { ... }
    out(native, foreign, length) { ... }
};
```

A `struct` statement in the transformation language declares a Linux structure and guides its transformation into a BSD/OS structure (or the reverse). Structure members are declared like they are in C. A structure member whose name begins with `linux_` is assumed to have no BSD/OS equivalent, and it doesn't get converted automatically. Unlike flags or cookies, structs have no defaults - all of the members must be listed, and if the BSD/OS version of the structure contains a member that is not present in the `struct` specification in the transformation language, **transform** assumes that no such member appears in the Linux version of the structure. Inside C escapes, the member names look exactly the way that they are declared - no prefixes are automatically prepended. (We can do this

because structure member names have a scope local to the given structure.)

When converting structures, each member is converted using a transformation that is appropriate for the type of the member, or if no transformation for that type is available, it is copied by assignment. Arrays are always copied by assignment (actually, by a `memcpy()` call). It is important to note that **transform** doesn't transform structures, but rather structure *pointers*; the direction and size of the transformation are derived from context.

After the specific members have been converted, any transformation functions are applied. The parameters to the transformation functions are a pointer to the source structure, a pointer to the destination structure and the length of the destination structure. If the last member in a structure is an array, **transform** assumes that the structure has variable length and it copies everything from the start of the array to the end of the structure as determined by the length parameter. Structure transformation functions have void type, since the parameters are passed by reference.

Here is an example of a `struct` statement:

```
struct sockaddr {
    familycookie_t sa_family;
    char sa_data[14];
    in(f, n, len) { n->sa_len = len; }
};
```

The `familycookie_t` type is a `cookie` type that converts socket family values from Linux numbers to BSD/OS numbers and back. Because the structure ends with an array, it is considered a variable-length structure and the `sa_data` field fills out the structure to the given length `len`. The input transformation fills in the BSD/OS `sa_len` field using `len`, whose value was supplied elsewhere.

Function

```
type-specifiers ... name(parameters, ...);
type-specifiers ... name(parameters, ...) =
    syscall-name;
type-specifiers ... name(parameters, ...) =
    errno-cookie;
type-specifiers ... name(parameters, ...) =
    number;
type-specifiers ... name(parameters, ...)
    { ... }
```

where *parameters* can be:

```
type-specifiers ... name
const type-specifiers ... name
volatile type-specifiers ... name
cookie-member-name
flag-member-name
```

A function statement is a transformation that converts a Linux function call into a BSD/OS function call. Function statements look similar to prototype function declarations and function definitions in C, but they have different meanings.

All of the function statement formats require a return type, a function name and a parameter list. The return type doesn't have to be a transformable type; it may be any C type, including a pointer, as long as all of the type names have been declared. The function name should match a name in the Linux C library. **Transform** uses a database of library symbols to generate all of the aliases for a known symbol, so the simplest version of the symbol name is usually the right one. If the name is identical to the name of a BSD/OS system call, the body of the function may be omitted, in which case **transform** arranges to call the BSD/OS system call automatically. There may be zero or more parameters. Each parameter is either a declaration for a name, or a cookie or flag member name. Declared parameters look much like they do in C, except that the name of the parameter is mandatory even when the function statement has no body and looks like a C declaration. Here is a simple example:

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

This definition creates a mapping for the `read()` function. It replaces the definitions for the Linux names `read`, `__read` and `__libc_read`. It calls BSD/OS system call number 3 (`SYS_read`) with the given parameters and returns the result. If the return value is -1, it converts the error number in `__bsdi_errno` into a Linux error number in `errno`.

The parameter list may optionally be followed by an assignment or a C escape. An assignment is a shorthand for certain common function bodies. An assignment from a system call name tells **transform** to make a call to the given BSD/OS system call rather than using the name of the function as the name of the system call. An assignment from an `errno` cookie says to return an error condition (-1 for integer valued functions, NULL for pointer valued functions) and set the Linux `errno` variable to the given value (translated to a Linux value). An assignment from a number tells **transform** to make the function return that constant value; it's useful for turning functions into no-ops. Finally, if you provide a C escape, it will be used as the body of the function in C. **Transform** will still look for error returns and translate `errno` unless you mark the function definition with the type qualifier `noerrno`.

The basic point of the transformation language is to allow you to specify transformations of function parameters and return values using transformable type names.

If the parameter and return value transformations are sufficient to handle the transformation for the function, then you can generally omit the function body; this is the simplest and most common definition in the transformation language. If you need to do more work, you can write your own function body. Inside the function body there are a few rules that you must follow, which (unfortunately) are not enforced by **transform**, which does not process the C code in the body. Functions must be re-entrant; if they need to allocate memory dynamically, they should do it on the stack using stack variables and/or the `alloca()` function. To make a BSD/OS system call inside a function body, you must do an indirect call through the `__bsdi_syscall()` function. The `__bsdi_syscall()` function works just like the BSD/OS `syscall()` function - it takes a syscall number from `<sys/syscall.h>` and a list of parameters, and it performs the corresponding BSD/OS system call. Any helper functions that you provide in C escapes must have names that use the `__bsdi_` prefix so that they do not collide with Linux function names. The `__bsdi_syscall()` function sets the `__bsdi_errno` variable, *not* the `errno` variable, which is a Linux variable. **Transform** generates code to translate `__bsdi_errno` to `errno` automatically, so in general it isn't necessary to refer to `__bsdi_errno` explicitly.

There are several interesting features of parameters beyond the obvious ones. Transformable structure pointers are quite special in many ways:

- The `const` keyword means something in addition to the usual C semantics when it is applied to a transformable structure pointer. A `const` transformable structure pointer is an input-only parameter - the structure gets converted from a Linux structure into a BSD/OS structure, copying it from the application's memory space onto the stack; however, no copying or condition is performed on return.
- A `volatile` transformable structure pointer is a read/write parameter - that is, it is transformed both on input and on output, unless there is an error.
- A transformable structure pointer parameter that doesn't have a `const` or `volatile` qualifier is output-only. The BSD/OS system call places its data in a BSD/OS structure allocated on the stack, and that structure is automatically converted on return into the corresponding Linux structure.
- If a structure definition ends with an array and a function definition contains both a transformable structure pointer *and* an integral parameter with whose name consists of the prefix `length_` plus the name of the structure pointer parameter, then the structure is considered to be variable length and it is assumed to have the number of bytes indicated by

the length parameter. The length parameter may also be a pointer to an integral type, in which case it is dereferenced before it is used. The value of the length parameter is used when converting the array member (as described above) and it is also passed to the structure's transformation functions, if it has any.

Some system calls like `ioctl()` and `fcntl()` change their parameter types or their return types depending on the value of a "command" or flag parameter. The transformation language allows you to define each of these variants separately. You simply specify a `cookie` or `flag` member name for a particular parameter, and if the function is called with that parameter matching that value, then the body of that function definition is executed. The first matching definition applies. You must always supply a *generic* function definition that uses the appropriate `cookie` or `flag` type for that parameter, and the body of that function definition is executed when the `cookie` or `flag` value fails to match any of the specific values in other function definitions for the same function. The feature is hard to describe in words but easy to show in examples; here's one:

```
cookie int linux_pers_t { PERS_LINUX 0; };
int personality(PERS_LINUX) = 0;
int personality(linux_pers_t p) = EINVAL;
```

This code defines a `cookie` that lists "personality" values for the Linux `personality()` system call. We only support the Linux personality, so only the `PERS_LINUX` member is interesting. If the application calls `personality(0)`, the definition for `personality(PERS_LINUX)` matches, and the system call appears to return 0. If the application calls `personality()` with any other value for the personality parameter, the system call will appear to return -1 and `errno` will be set to the Linux equivalent of `EINVAL`.

Here is a somewhat more complex example:

```
flag int openflags_t { ... };
cookie int fcntl_t { ... };
struct flock { ... };
openflags_t fcntl(int fd, F_GETFL, int ignore);
int fcntl(int fd, F_SETFL, openflags_t oflags);
int fcntl(int fd, F_GETLK, struct flock *fl);
int fcntl(int fd, F_SETLK,
    const struct flock *fl);
int fcntl(int fd, F_SETLKW,
    const struct flock *fl);
int fcntl(int fd, fcntl_t cmd, int arg);
```

In this example, if the second parameter matches `F_GETFL`, then the first `fcntl()` definition applies; it converts the BSD/OS `open()` mode flags into Linux mode flags on return. If the second parameter is `F_SETFL`, the third parameter is converted from Linux

mode flags into BSD/OS mode flags before we call the BSD/OS `fcntl()` system call. The third, fourth and fifth definitions show how a transformable structure pointer parameter is converted on output (third) and input (fourth and fifth, respectively). The generic function definition causes all remaining `fcntl()` `cookie` values to be passed unchanged to the BSD/OS `fcntl()` system call; this is appropriate when the BSD/OS `cookie` value is identical to the Linux `cookie` value and the parameters and return value do not require transformation, or when the application supplies an illegal `cookie` value that the BSD/OS `fcntl()` call can reject. Note that there is really just one `fcntl()` function in the **liblinux** library - all the `fcntl()` definitions are merged into a single function.

The `__kernel_` feature

Function names that begin with `__kernel_` are treated specially. LAP has support for raw Linux system call traps. By default, when it detects a Linux system call trap, LAP marshals its arguments and transfers control to the function with the same name as the Linux kernel call. Sometimes it isn't appropriate to do this - for example, the kernel system call may have a different name from the function in the Linux C library, or it may treat its parameters differently. In that case, you may define a function with the `__kernel_` prefix to handle just system call traps.

For example, the Linux `llseek()` system call has a different kernel interface from the Linux C library interface:

```
linux_loff_t llseek(int fd, linux_loff_t offset,
    int whence) { ... }
int __kernel_llseek(int fd, unsigned long o_high,
    unsigned long o_low, linux_loff_t *result,
    int whence) { ... }
```

The kernel version of this function swaps the high and low words of the offset and returns its value using a reference parameter, unlike the C library version.

5. The implementation

5.1. The transformation compiler

The **transform** program compiles transformation sources and creates C output files. The program is about 3,300 lines of C, Yacc and Lex source code. An additional program named **afdb** builds databases for **transform**; it's about 250 lines of C.

The **transform** program is a single-pass compiler. It parses statements and then emits most code in place, including C escapes. Typedef and struct declarations are converted into C typedef and struct declarations, respectively, while `cookie` and `flag` members become C `#define` directives. `In()` and

`out()` transformation functions become static inline functions with names that begin with the type name (for `typedef` types), with the type name plus `_default` (for cookie or flag types) or with the tag name (for structures); the function names end in `_in()` and `_out()`, respectively. Cookie and flag transformations turn into `switch` statements or sequences of `if` statements (respectively), inside inline functions, with a call to the `type_default_in()` or `type_default_out()` function at the end, as appropriate. Function statements become C inline functions with numeric suffixes to distinguish the different alternatives. At the end of processing, the compiler emits static functions which test incoming arguments and call the appropriate function alternatives. The compiler also generates assembly escapes that serve to map the static container functions onto the names that the Linux C library uses. (The inline function and assembly escape syntax are extensions in the GNU C compiler.)

The `afdb` program processes the dynamic symbol table from the Linux C library and produces Berkeley DB btree files that map addresses to function names and function names to addresses. When the transformation compiler sees a function name, it looks the name up in the database, locates all of the aliases, then generates assembly escapes that duplicate the Linux aliases in the emulation library.

5.2. The transformations

Currently there are about 3250 lines of code written in the transformation language.

Most of the transformations are straightforward. Many functions require only the default transformation, with no transformable parameters or transformable result and no function body; in that case we just make the equivalent BSD/OS system call, and transform the BSD/OS `errno` value if there is an error. A number of functions apply very simple transformations on parameters. Some functions perform minor API changes; for example, Linux has two kernel interfaces for the `select()` function, one of which takes different parameters from the standard API, and one of which has a different name, and the transformation language serves to map parameters correctly.

A few transformations are more complex. The `ioctl()` transformations are large because Linux `ioctl()` cookies and `termio/termios` structures are different from BSD/OS, even though the semantics are very similar. For socket functions, the Linux kernel provides a single system call that multiplexes all of the BSD-style socket calls using cookies, so the socket support is a little bit complicated. The `getdirentries()` and `getdents()` transformations are

complicated because Linux `dirent` structures have seek-offset members that are not in BSD/OS `dirent` structures, and because the different sizes of the Linux and BSD/OS `dirent` structures require code to re-pack them. A similar issue with re-packing applies to `getgroups()` and `setgroups()`, which require a separate kernel implementation because the Linux kernel `gid_t` data type is a 16-bit integer while the BSD/OS type is a 32-bit integer. (The problem doesn't strike the C library API for `getgroups()` and `setgroups()` because the GNU C library that Linux uses has a 32-bit `uid_t` type like BSD/OS. There are a few cases like this where the GNU C library API is closer to the BSD/OS API than to the Linux kernel API and we try to take advantage of this when we can.)

5.3. The libraries

We build three shared library objects: the dynamic linker, the emulation library and a dummy C library.

The *dynamic linker* is built from the GNU C library source code and linked with transformation language source code so that it can make native BSD/OS system calls. We configure the dynamic linker slightly differently from Linux so that it looks for its `ld.so.cache` file in `/linux/etc` rather than `/etc`, which causes it to use different libraries from the native BSD/OS dynamic linker.

We make the *emulation library* from the transformation sources plus some assembly and C code. We supply code to do call tracing at the Linux API level. We add code to implement BSD/OS system call stubs without polluting the Linux C library namespace. We add initialization code that programs the hardware interrupt descriptor table so that Linux system call interrupts are dispatched to an address in the emulation library, and we generate a dispatch table that sends system call interrupts to the appropriate handler. The transformation compiler itself generates stub code for each system call that marshals arguments and calls the C transformation function; the dispatch table jumps to the stubs. We build the dispatch table using an `awk` script that processes the Linux header file that defines Linux system call numbers.

The *dummy C library* replaces the Linux C library in the Linux library path. The purpose of the dummy C library is to load the emulation library ahead of the real Linux C library in the symbol search path, so that no matter how an application tries to load the Linux C library, it will always get the emulation library too. We use the `ELF DT_AUXILIARY` feature to implement this trick. The dummy C library would not have any code of its own if it were not for a peculiar rule about library initialization. It seems that the dynamic linker

initializes libraries in reverse order of their loading; that means that it initializes the emulation library after it initializes the real Linux C library. But we have to arrange to dispatch Linux system call interrupts before we can execute any code from the real Linux C library, so the emulation library needs to run its initialization first. The dummy C library initialization is performed before both the real Linux C library initialization and the emulation library initialization, however, so we get around this problem by arranging for the dummy C library to call an initialization function in the emulation library.

5.4. How does it really work?

It's hard to tell how the emulation really works just by reading descriptions of its pieces. Here's a brief description of what happens when you actually run a program.

When a Linux application starts up on BSD/OS, the normal **ELF** loader in the operating system loads it with the modified Linux dynamic linker. The application uses the dynamic linker to load the shared libraries that it needs. All of the shared libraries that it sees are real Linux shared libraries, with one exception: when the application asks for the Linux C library, it also gets an emulation library.

Let's say the application needs a service from the Linux C library; for example, it does a `stat()` call to find out the size of a file. The application doesn't define `stat()` itself, so the dynamic linker looks for an implementation of `stat()`. Because of the way the libraries were loaded, the dynamic linker looks in the emulation library before it looks in the Linux C library, and it uses the emulation library's `stat()` function. The emulation function allocates room on the stack for a BSD/OS `stat` structure and calls the BSD/OS `stat()` function with the stack buffer as an argument. If the BSD/OS `stat()` call succeeds, the emulation function copies and converts the elements of the buffer into the Linux `stat` structure that was passed in and returns 0 for success. If the `stat()` call failed, the emulation function converts the BSD/OS error number into a Linux error number and stores the result in the Linux `errno` location.

Current Linux shared C libraries are statically linked internally, so a `stat()` call inside the C library works a little differently. The Linux C library moves the system call number (106) and the two parameters into registers and executes the Intel `int $0x80` instruction to generate a software interrupt. The Intel hardware transfers control directly to a dispatch routine in the emulation library. The dispatch code performs a computed `goto` using the system call number, resulting in a branch to the automatically generated stub for `stat()` in the

emulation library. This code pushes the parameters on the stack and calls the same `stat()` emulation function that the application used in the example above. On return, if there was an error, the stub code copies the negated `errno` value back into the result register.

(This example oversimplifies the specific situation with `stat()` slightly - see the appendix for more details.)

6. Conclusions

6.1. Comparisons to other work

Of course there are many ways to emulate other operating systems and Linux on other Unix-like systems in particular. I want to mention a couple other emulations done in a different style, and compare them to LAP.

- The Skunkworks folks at **SCO** have a very neat emulator that they call **lxrun**. **Lxrun** is a program that loads a Linux program into its address space and catches the `SIGSEGV` signal that the SCO Unix operating system sends to the program when the Linux code executes a software interrupt instruction (`int $0x80`). This is analogous to the way that LAP redirects the hardware interrupt descriptor table, but it uses unprivileged software instead, so it requires no changes to the kernel at all, although it's a little slower. **Lxrun** can handle statically linked programs as well as the obsolete Linux **a.out** executable format, unlike LAP, and it requires no changes at all to the dynamic linker and no futzing with libraries. It's a really lightweight implementation. LAP improves on it by reducing the overhead of software interrupts, reducing overhead again by interposing the library interface to system calls when possible, and by loading itself automatically rather than requiring a separate loader program (at the expense of modifying the dynamic linker to use BSD/OS system calls). LAP's transformation language should also make it easier to maintain.
- The **FreeBSD** project decided to implement Linux emulation in its kernel. All of the transformations are performed in privileged mode, and the memory for the emulation is dedicated. This is a heavyweight implementation in terms of the amount of code required and its effect on the kernel, but it does permit precise emulation of (for example) signal semantics. While this is nice, I feel that operating system kernels are already absurdly fat, and given that LAP can be reasonably complete and efficient operating outside the kernel, that's a virtue. Certainly the transformation-driven approach to emulation could be applied to an in-kernel emulation if we felt that it would be useful.

Another out-of-kernel approach that we could have taken was the microkernel plus OS server approach that

was used in Mach 3 and later versions of Mach [Golu90]. In Mach, not just the libraries but entire machine-independent part of the operating system runs outside the kernel, and the emulation communicates with the kernel using IPC calls. That strategy would clearly be overkill for a Linux emulation on a BSD Unix system, however, since Linux and BSD are so similar at the API level.

It's worth pointing out that while the transformation compiler was written in such a way to make it easy to retarget for (say) FreeBSD, I would not consider it a flaw if it were never retargeted. It's nice to be able to generalize tools, but LAP benefits from using a little language regardless of whether it is general.

6.2. General results

The emulation is quite successful. We can run a number of interesting Linux applications, and they run quite efficiently. Among the programs we have tested are the Adobe Acrobat Reader v4, Netscape Communicator v4.7, and WordPerfect v8. (In fact, this document was composed using Netscape Composer for Linux running under BSD/OS.) Only very minor BSD/OS kernel modifications were required, and the kernel contains no emulation code itself, so we avoided any significant kernel bloat. The implementation is remarkably robust so far; we have had to make very few bug fixes after the initial coding and testing. I attribute this to the small size and the simplicity of the specification.

As far as performance goes, LAP seems to be more than adequate, but the impact is difficult to measure in a meaningful way. I thought about trying to measure some application running under native Linux and comparing it to the same application running under LAP on native BSD/OS on the same hardware, but the different kernels, filesystems and other factors would surely confound the result - it would be more of a measure of Linux versus BSD than the overhead of LAP. However, I can provide a vague idea of how much time is spent in LAP when running a program. I ran an instruction tracer on the Linux `ls` program running `ls -l` under LAP and counted the number of instructions that were executed in the range of addresses occupied by LAP. For an 8-item directory listing, LAP used 3423 instructions out of 714797 total, or about 0.5%. Because of the way that LAP interposes itself in front of the Linux C library, the impact is actually a bit less than it seems, because LAP replaced code that would have been executed in the `ls` program under Linux. When running an X-based program under LAP such as Netscape (as I am doing right now), the overhead is not perceptible.

6.3. Bugs, omissions and other niceties

Not all Linux system calls are currently emulated. The sign that a system call hasn't been emulated is that your application prints a message and aborts. Almost all of the missing system calls are administrative calls, however, so we suspect that we won't encounter them in third-party applications. (For example, we don't support Linux NFS daemons; the BSD/OS native NFS daemons work just fine.)

By far the biggest user-visible omission is the lack of support for the Linux `clone()` system call and related user thread support. I am working actively on this issue and I hope to have news to report at the conference.

We don't support statically linked Linux programs. If we wanted to support statically linked Linux programs, we would adopt the SCO emulation technology. The kernel would load a statically-linked version of the emulation library into every statically linked Linux program. The emulation library would not interpose itself in front of Linux C library functions, but it would still catch software interrupts and process them in the same way that LAP currently does. Statically linked Linux programs are sufficiently rare that we have not seen a need for this feature yet.

Currently LAP does not do any mapping of data types that have narrower widths on Linux. If a UID on BSD is greater than 65,536, a LAP program may see a truncated value for that UID instead of the full 32-bit value in some situations.

I have not described the LAP support for older Linux **ELF** programs that use the 5th version of the Linux C library (sometimes called **libc5**). We do support those programs and the **libc5** emulation shares most of its source code with the rest of LAP. It works a little differently from the scheme described in this document, but I'm not going to explain it here. LAP does not (and probably will never) support Linux **a.out** programs.

The code for LAP is available on the BSD/OS contributed software CD-ROM. Like all software on that CD-ROM, it is freely redistributable. Feel free to use it and modify it as you please with the usual understanding that if it doesn't work for you or causes problems for you, BSDI doesn't take any responsibility.

Appendix - an extended example

types.xh:

```
/*      BSDI $Id: types.xh,v 1.2 1999/04/14 22:42:57 prb Exp $ */

/*
 * Linux has an awkward problem which we have to worry about here...
 *
 * The GNU C library defines basic types that don't match Linux kernel types.
 * The library applies transformations to these types when passing them
 * to and from the Linux kernel. By convention, the transformation routines
 * call stubs named __syscall_*() to perform the syscall using Linux
 * kernel data types. It's stupid to transform these data types twice,
 * especially when the GNU C data types are generally wider and hence
 * closer to BSD types, so we interpose our transformation routines over
 * the GNU C library routines rather than the Linux __syscall_*() routines.
 */

%{
#include <sys/types.h>
%}

/*
 * Here are the GNU C library types.
 */
typedef unsigned long long dev_t {
    in(dev) { return (makedev(dev>> 8, dev & 0xff)); }
    out(dev) { return (major(dev) << 8 | minor(dev)); }
};

typedef char *caddr_t;
typedef long clock_t;
typedef unsigned int gid_t;
typedef unsigned long ino_t;
typedef int key_t;
typedef long long linux_loff_t;
typedef unsigned int mode_t;
typedef unsigned int nlink_t;
typedef long off_t;
typedef int pid_t;
typedef int ptrdiff_t;
typedef unsigned int size_t;
typedef int ssize_t;
typedef long time_t;
typedef unsigned int uid_t;

/*
 * For documentation purposes, here are the actual Linux internal types,
 * where they differ from the GNU C library types.
 */

typedef unsigned short linux_kernel_dev_t;
typedef unsigned short linux_kernel_gid_t;
typedef unsigned short linux_kernel_mode_t;
typedef unsigned short linux_kernel_nlink_t;
typedef unsigned short linux_kernel_uid_t;

typedef unsigned int u_int;
typedef unsigned short u_short;
typedef unsigned long u_long;
```

stat.xh:

```
/*      BSDI $Id: stat.xh,v 1.2 1999/04/14 22:38:59 prb Exp $ */

/*
 * Transforms for stat.h.
 */
```

```

%{
/* Don't use timespecs. */
#define _POSIX_SOURCE 1
#include <sys/stat.h>
#undef _POSIX_SOURCE
#define old_stat stat
%}

cookie int linux_stat_ver_t {
    _STAT_VER_LINUX_OLD 1;
    _STAT_VER_SVR4 2;
    _STAT_VER_LINUX 3;
};

struct stat {
    dev_t st_dev;
    unsigned short linux_pad1;
    ino_t st_ino;
    mode_t st_mode;
    nlink_t st_nlink;
    uid_t st_uid;
    gid_t st_gid;
    dev_t st_rdev;
    unsigned short linux_pad2;
    off_t st_size;
    unsigned long st_blksize;
    unsigned long st_blocks;
    time_t st_atime;
    long linux_unused1;
    time_t st_mtime;
    long linux_unused2;
    time_t st_ctime;
    long linux_unused3;
    long linux_unused4;
    long linux_unused5;
};

struct old_stat {
    unsigned short int st_dev;
    unsigned short int linux_pad1;
    unsigned long int st_ino;
    unsigned short int st_mode;
    unsigned short int st_nlink;
    unsigned short int st_uid;
    unsigned short int st_gid;
    unsigned short int st_rdev;
    unsigned short int linux_pad2;
    unsigned long int st_size;
    unsigned long int st_blksize;
    unsigned long int st_blocks;
    unsigned long int st_atime;
    unsigned long int linux_unused1;
    unsigned long int st_mtime;
    unsigned long int linux_unused2;
    unsigned long int st_ctime;
    unsigned long int linux_unused3;
    unsigned long int linux_unused4;
    unsigned long int linux_unused5;
};

stat.x:
/*      BSDI $Id: stat.x,v 1.2 1999/04/14 22:47:01 prb Exp $      */

/*
 * Transformation rules for <sys/stat.h> syscalls.
 */

```

```

include "types.xh"
include "stat.xh"

/*
 * Linux's kernel types don't match its user types.
 * The GNU C library performs transformations from the kernel types
 * to the user types. We interpose the GNU C library stubs rather
 * than the Linux kernel stubs, so that we don't transform twice.
 */
int __syscall_stat(const char *name, struct old_stat *buf)
{
    return (__bsdi_syscall(SYS_stat, name, buf));
}

int _xstat(_STAT_VER_LINUX_OLD, const char *name, struct old_stat *buf)
{
    return (__bsdi_syscall(SYS_stat, name, buf));
}
int _xstat(_STAT_VER_LINUX, const char *name, struct stat *buf)
{
    return (__bsdi_syscall(SYS_stat, name, buf));
}
int _xstat(linux_stat_ver_t v, const char *name, struct stat *buf) = EINVAL;

int _fxstat(_STAT_VER_LINUX_OLD, int fd, struct old_stat *buf)
{
    return (__bsdi_syscall(SYS_fstat, fd, buf));
}
int _fxstat(_STAT_VER_LINUX, int fd, struct stat *buf)
{
    return (__bsdi_syscall(SYS_fstat, fd, buf));
}
int _fxstat(linux_stat_ver_t v, int fd, struct stat *buf) = EINVAL;

int _lxstat(_STAT_VER_LINUX_OLD, const char *name, struct old_stat *buf)
{
    return (__bsdi_syscall(SYS_lstat, name, buf));
}
int _lxstat(_STAT_VER_LINUX, const char *name, struct stat *buf)
{
    return (__bsdi_syscall(SYS_lstat, name, buf));
}
int _lxstat(linux_stat_ver_t v, const char *name, struct stat *buf) = EINVAL;

/*
 * The other stat.h calls...
 */

int chmod(const char *path, mode_t mode);

int fchmod(int fd, mode_t mode);

int mkdir(const char *path, mode_t mode);

int umask(mode_t m);

```

References

- [ANSI89] American National Standard for Information Systems, *Programming Language - C*, ANSI X3.159-1989.
- [BSDI00] <http://www.bsd.com/>
- [Free00] *Using and Porting GNU CC, for Version 2.95*, R. Stallman, Free Software Foundation, 2000.
- [Ging89] "Shared Libraries in SunOS," R. Gingell, M. Lee, X. Dang, M. Weeks, in *Proceedings of the Summer 1989 Usenix Conference*, 1989.
- [Golu90] "Unix as an application program," D. Golub, R. Dean, A. Forin, R. Rashid, in *Proceedings of the Summer 1990 Usenix Conference*, June 1990.

- [IEEE96] IEEE Std 1003.1, 1996 Edition, *Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C language]*, Institute for Electrical and Electronics Engineers, 1996.
- [John75] "Yacc - Yet Another Compiler Compiler," S. Johnson, *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill NJ, 1975.
- [John79] "A Tour through the Portable C Compiler," S. Johnson, *Unix Programmer's Manual*, 7th Edition, volume 2b, AT&T Bell Laboratories, Murray Hill NJ, 1979.
- [Lesk75] "Lex - a lexical analyzer generator," M. Lesk, *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill NJ, 1975.
- [Linu00] <http://www.linux.org/>
- [Olso99] "Berkeley DB," M. Olson, K. Bostic, M. Seltzer, in *Proceedings of the Freenix Track*, 1999 Usenix Annual Technical Conference, June 1999.
- [Reco98] "Linux Emulation for SCO," R. Record, M. Hopkirk, S. Ginzburg, in *Proceedings of the Usenix 1998 Annual Technical Conference: Invited Talks and Freenix Track*, June 1998.
- [Ritc79] "A Tour through the UNIX C Compiler," D. Ritchie, *Unix Programmer's Manual*, 7th Edition, volume 2b, AT&T Bell Laboratories, Murray Hill NJ, 1979.
- [Salu98] *Handbook of Programming Languages, Volume III: Little Languages and Tools*. P. Salus, ed. Macmillan Technical Publishing, 1998.
- [Unix90] *System V Application Binary Interface*, Unix Software Operation, Prentice-Hall, 1990.

Traffic Data Repository at the WIDE Project

Kenjiro Cho
Sony CSL

kjc@csl.sony.co.jp

Koushirou Mitsuya
Keio University

mitsuya@sfc.wide.ad.jp

Akira Kato
The University of Tokyo

kato@wide.ad.jp

Abstract

It becomes increasingly important for both network researchers and operators to know the trend of network traffic and to find anomaly in their network traffic. This paper describes an on-going effort within the WIDE project to collect a set of free tools to build a traffic data repository containing detailed information of our backbone traffic. Traffic traces are collected by *tcpdump* and, after removing privacy information, the traces are made open to the public. We review the issues on user privacy, and then, the tools used to build the WIDE traffic repository. We will report the current status and findings in the early stage of our IPv6 deployment.

1 Introduction

In this paper, we introduce an on-going effort within the WIDE project to collect a set of free tools to build a traffic data repository containing detailed information of our backbone traffic. The WIDE project makes the resulting data sets publicly accessible so that this project is not only on freely-redistributable software but also on freely-redistributable traffic data sets.

The WIDE project is a research consortium in Japan established in 1987. The members of the project include network researchers, engineers and students of universities, industries and government. The focus of the project is empirical study on a live large-scale internet. Thus, WIDE runs its own internet testbed carrying both commodity traffic and research experiments. WIDE is also responsible for various Internet operations including the M-root name server, NSPIX(P(Network Service Provider Internet eXchange Point), AI3(Asian Internet Interconnection Initiatives), and 6Bone in Japan.

The goals of our traffic repository are to promote traffic analysis research as well as to promote development of tools. Traffic characteristics in a backbone network are considerably different from those in a local area network but few people have access to traffic traces from back-

bone networks. Obtaining details of backbone traffic is getting harder as more backbone networks are shifting to commercial ISPs, which motivate us to build a traffic repository [KMKA99]. Traffic traces from the 6Bone is also made available to promote development of IPv6-ready tools.

Traffic traces are collected at several points within the WIDE backbone. Traces are in the *tcpdump* raw format so that all header information is available and can be used for detailed analysis.

We use commodity hardware and the existing freely-available tools for building our traffic repository so that it has nothing technically fancy. Our focus is rather continuity in making the latest traces available. At this writing, daily traces at one sample point have added up to the record of more than a year.

2 Related Work

Packet Monitoring

Packet capturing was brought with the advent of Ethernet. The first personal computer, Xerox Alto, already had programs to monitor Ethernet. As Ethernet came into wide use, dedicated network monitors became indispensable to developers and operators. The CMU/Stanford enet packet filter is the first UNIX based packet filter developed in 1980 [MRA87]. It eventually evolved into the Ultrix Packet Filter at DEC, NIT under SunOS, and BPF.

Userland programs that prints the headers of packets appeared with UNIX workstation. Sun implemented NIT (Network Interface Tap) to capture packets and *etherfind* to print packet headers. The advantage of UNIX-based monitoring tools is that users can use other software tools available on UNIX for manipulating and analyzing packet traces.

tcpdump [JLM89] is probably the most popular packet capturing tool in the UNIX community. *tcpdump* first appeared in 1989 and merged into BSD Net Release2 in 1991. *tcpdump* is based on a powerful filtering mecha-

nism, the BSD packet filter (BPF) [MJ93]. The packet capturing and filtering facilities of *tcpdump* are implemented in a separate library, *pcap* [JLM94]. The *pcap* library became independent from *tcpdump* in 1994, and there are a wide range of network monitoring or analysis tools which integrate the *pcap* library. In 1999, *tcpdump.org* [tcp99] was organized by volunteers to maintain the *tcpdump* code.

High-performance monitoring systems are explored by OC3MON [ACTW96] and its successors that are based on a PC hardware but exclusively for ATM. Coral-Reef [CAI99] is a package developed at CAIDA to analyze the output of OCxMON.

Packet monitoring techniques have been used to gather long-term statistics. A pioneering work is *statspy* [Bra88] in the NNStat package developed at ISI. As SNMP becomes widely available, network statistics tools are geared toward SNMP. MRTG [Oet96] and its successor RRDtool [Oet99] are popular tools to collect traffic counters from routers through SNMP. More recently, *eflowd* [McR99] is developed at CAIDA to make use of Cisco's NetFlow [Cis98] that exports statistics of flow cache entries.

Traffic Archive

The Internet Traffic Archive (ITA) [DMPS95] was created in 1995 by Danzig *et al.* to promote research on network analysis. ITA has several traces studied in published papers as well as unstudied traces. ITA is an important step towards open traffic data sets because research based on open data sets can be confirmed or further analyzed by someone else, which leads to deeper studies.

There are several different formats in the ITA archives but the majority of the available traces are in the *tcpdump* ascii output format. A set of shell scripts, called *sanitize*, are written by Paxson and used to scramble addresses in the *tcpdump* ascii output format to provide anonymity to network users.

Our traffic repository was motivated in part by the effort at ITA. We employ automatic traffic sampling at regular intervals since the archives at ITA are not updated much. We also thought that the *tcpdump* raw format is preferable to the ascii format because the raw format has more information, and powerful tools are available to manipulate the raw format. Also, it would be useful for developers of tools which handle traces in the *tcpdump* raw format.

3 Motivation

WIDE installed several traffic sampling points within the backbone since traffic data has been essential to both

network research and operation. However, traffic information tend to be confined to a small set of members, and it is difficult to share detailed information without a framework to support sharing. This leads to the idea of a traffic trace repository in which detailed traffic traces are archived and easily accessible to everyone.

In order to build a traffic trace repository and make good use of traces, we had to solve two problems. One is to create a safety measure for handling traces that include privacy information. The other is automation of the trace acquisition process.

Traffic traces include private information of the network users. Special care is needed to handle traces, and thus, only limited members are allowed to handle raw traces. Still, there is always a risk of accidents when we handle raw traces. Hence, even if traces are available only for limited members, it is important to make traces safe enough to prevent possible accidents. On the other hand, if traces are made free from user privacy, we can make the traces open to the public since WIDE does not need to worry about its impact to stock prices.

Automation of the maintenance process is the other important factor. Collecting traffic traces in a long term needs perseverance, and cannot be achieved unless most of the work are automated. Not only automation of acquisition but also automation of summarization and visualization are essential to maintaining the repository because, if no feedback is given, people tend to run out of energy.

There are strong concerns about security and privacy with regard to making traces publicly available. After a long discussion, we have reached a conclusion that the benefits outweigh the risks. Or, at least, it is worth a challenge.

4 Privacy Issues

Traffic traces contain privacy information including network addresses and application payload so that it is important to understand issues involved in user privacy.

There are two major issues involving user privacy.

Removing user data: User private data must be removed from traces. Traffic traces should have only protocol headers, Protocol payload which contains user data should be removed.

Providing anonymity: IP address is unique and can be used to identify a user, and thus, addresses should be scrambled to provide anonymity to users.

There are a wide variety of research purposes that have different requirements for traces. No single method will satisfy all the requirements and still keep user privacy. We are trying to provide traces which can be used for a wide range of research. For research which has specific

requirements, our traces will provide a starting point, and can be used to narrow down its requirements. Once specific requirements become clear, it is easy to find a specific method to meet the requirements.

4.1 Removal of Payload

As a general rule, we should remove the payload of TCP or UDP that contains users' private information. If another protocol header exists on top of a TCP or UDP header and the inner header does not contain user private information, the inner header may be maintained. If it is difficult to judge whether a header contains user private information or not, the header should be removed as a precaution.

Once protocol payload is removed, the risk of jeopardizing user privacy is considerably reduced. It would be safe enough for use within a closed group. However, in order to make traces open to the public, we need a further level of security. That is, we need to provide anonymity to network users.

4.2 Address Scrambling

We should provide anonymity to individuals and organizations by scrambling source and destination addresses in IP headers. IP addresses, however, have hierarchical structures and special addresses such as broadcast addresses, multicast addresses and private addresses. It is not easy to provide anonymity but still keeping the structures and special meanings. We should choose an appropriate method according to the importance of anonymity in traces and the purpose of the data set.

Address Scrambling Methods

Address scrambling maps one IP address to another IP address. There are a number of methods to scramble addresses.

1. the sequential numbering method maps each IP address occurrence to a sequential number. Although this method is easy to understand, it is difficult to preserve other meanings of addresses.
2. the hash method maps an IP address to another IP address using a hash function in order to provide random mapping. It is also possible to preserve the common address prefix between 2 addresses by maintaining an ordered tree of addresses similar to a routing table. In this method, if 2 IP addresses have a common address prefix, they are mapped to addresses with a common address prefix of the same length. Note that, although it preserves routing information, this method has a risk of being reverse-engineered. For example, one can use a well-known server's address as a clue to de-scramble the address

prefix [Ylo96]. The impact of this threat, however, depends on the importance of hiding the network topology.

There are several choices regarding address consistency between two or more data sets.

1. all occurrences of an address are to be mapped to a single address within a data set.
2. all occurrences of an address are to be mapped to a single address across different data sets.

Longer consistency is convenient for analysis but it also makes reverse-engineering easier.

Address Issues

non-unique addresses Addresses not containing user identifiers may be left without scrambling. Those addresses include broadcast addresses, multicast addresses, and private addresses. In the case of IPv6, link-local addresses and site-local addresses could contain unique interface identifier (e.g., MAC address). A solicited-node multicast address contains lower bits of the global address. Therefore, these IPv6 addresses should be scrambled as well.

addresses in upper layers IP addresses could be contained in an upper protocol message. For instance, ICMP and DNS contain IP addresses in their protocol payload. These addresses must be scrambled in the same manner, or removed.

MAC addresses Link-layer headers (e.g., Ethernet headers) contain MAC addresses. A MAC address contains vendor and model information which could be part of user privacy or lead to a security hole. However, traces from backbone networks do not contain MAC addresses of user nodes since MAC addresses recorded in the trace are only from local nodes on the same segment.

IP/TCP options IP options can contain IP addresses. Addresses in IP options should be scrambled in the same manner. Otherwise, IP options should be replaced by NOP options, or removed.

On the other hand, TCP options do not contain privacy information. TCP options carry useful information to analyze TCP behaviors so that TCP options may be preserved.

5 Methods

We use several tools to automatically maintain the traffic repository. The details of these tools are described later in this section. New trace data is collected from sampling

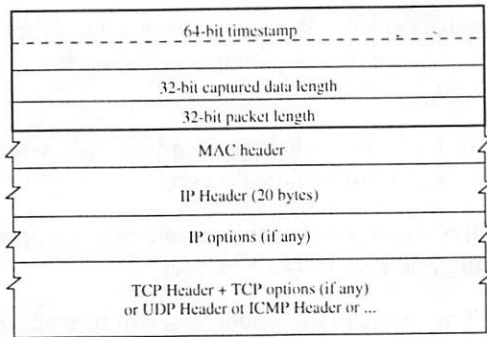


Figure 1: Pcap header format

points to the repository during the night. A web page for the new trace is automatically created.

At a sampling node, a script is invoked from *cron* to run *tcpdump* and compress the trace. The obtained raw trace file is placed under a certain directory.

At the repository node, another script is invoked from *cron* to fetch the raw trace and process it. The script copies the compressed raw trace from the sampling point over a secure session using *scp*. Then, the script uncompresses the trace and invokes *tcpdpriv* to remove privacy information from the trace. The trace is fed into *tcpdstat* to get a summary output. The script creates a web page for the trace, and updates the index page to include the newly created page. Finally, the script compresses the trace data again, and place it for the ftp service.

5.1 tcpdump

We use *tcpdump* to obtain traffic traces because *tcpdump* is widely used, and installed as part of the default tools on many systems. In addition, there are many tools that integrates the *pcap* library and be able to read *tcpdump* output files. Those tools include *tcptrace*, *tcpslice*, *tcpdstat* and *ttt*.

tcpdump, by default, puts the network interface into *promiscuous mode* to capture every packet going across the wire. In the BSD-derived kernel, BPF is implemented as a packet capture mechanism. When BPF is enabled, the network driver in the kernel passes both sending and receiving data-link level frames to BPF. BPF performs packet filtering if necessary, adds timestamp, and copies the fixed length from the head of the frame into the store buffer. *tcpdump* in the user space can read multiple frames in a single read from the store buffer in the kernel in an efficient manner. *tcpdump*, by default, prints the header information of each packet in a text format. With *-w* option, *tcpdump* writes out the packet frames into a specified file. With *-r* option, *tcpdump* reads from a saved file instead of a network interface to replay a saved file. The *pcap* library is used to read or write data in the

raw format. Thus, it is easy to write a program to read or write packets in the *tcpdump* format.

Figure 1 shows the format of raw *tcpdump* output. In the BSD systems, the kernel uses *microtime()* for timestamp; the precision of the timestamp is 1 usec on the PC architecture. Timestamp is taken when a packet is passed to BPF from the network driver so that it is the time that the driver sees that packet.

5.2 tcpdpriv

We use *tcpdpriv* to remove user data and scramble addresses. *tcpdpriv* was developed by Greg Minshall at Ipsilon Networks in 1996. *tcpdpriv* removes privacy information in a raw *tcpdump* output. *tcpdpriv* uses the *pcap* library to read and write *tcpdump* output files. *tcpdpriv* removes the payload of TCP and UDP, and the entire IP payload for other protocols. *tcpdpriv* implements several address scrambling methods; the sequential numbering method and its variants, and a hash method with preserving address prefix.

However, the original *tcpdpriv* lacks several features we need:

- it does not support IPv6.
- it does not preserve TCP options that are essential to analyzing TCP behaviors.
- we also want to preserve other protocols such as ICMP, ARP and DNS.

Thus, we have modified the original *tcpdpriv* to support these features. The default settings are also changed to meet our requirements since the options seem to be too complex and a mistake of option selection could be fatal to user privacy.

5.3 tcpdstat

We developed *tcpdstat* to get summary information of a *tcpdump* file. *tcpdstat* reads a *tcpdump* file using the *pcap* library and prints the statistics of a trace. The output includes the number of packets, the average rate and its standard deviation, the number of unique source and destination address pairs, and the breakdown of protocols.

tcpdstat is intended to provide a rough idea of the trace content. The output can be easily converted to a HTTP format. It also provides helpful information to find anomaly in a trace. For example, if the traffic volume of ICMP is unusually large, or if the traffic volume of a specific address pair is unusually large, it could be a sign of some form of a DoS attack.

5.4 Other Tools

There are other tools that are not used to create the traffic repository but can read *tcpdump* files and useful for analyzing traces afterwards.

tcpslice by Vern Paxson extracts portions of a trace. *tcptrace* by Shawn Ostermann produces detailed information about each TCP connection in a trace. *tracelook* by Greg Minshall provides *xgraph* plots of TCP connections in a trace. *flstats* also by Minshall prints flow statistics. *ethereal* by Gerald Combs is a traffic analyzer with a graphical user interface. *ethereal* uses the *pcap* library, and thus, can replay a *tcpdump* file. Our *ttt* (Tele Traffic Tapper) tool displays composition graphs of protocols and host addresses in real time. *ttt* can replay a trace file at a given speed so that it is possible to replay a 1-hour trace in 1 minute.

6 Current Status

Currently, we are collecting daily-traces from the following sampling points.

trans-pacific is a 1.5Mbps T1 line, one of the several international links of WIDE. The sampling point is on an Ethernet segment one hop before the T1 line. The incoming traffic (from U.S. to Japan) of this link is fairly congested.

6Bone is located on a FastEthernet segment connected to NXPIX-6 (An IPv6 internet exchange point in Tokyo) [WID99]. The segment located at an AS boundary, and the traffic includes only native IPv6 and does not include IPv4 except tunneled IPv4 over IPv6. Because NXPIX-6 is built on a FastEthernet switch, only the traffic crossing a single port of the switch can be captured.

Traces are sampled at a fixed time of day. This is obviously not desirable and we need to find a better sampling method.

We started daily data collection at the *trans-pacific* point in February 1999. Since WIDE has a number of connections to the Internet exchange points, the return path of a session does not necessarily go through the same link.

The maximum size of each trace file is limited to about 100M bytes (about 40MB when compressed). We believe 100MB is an appropriate size for handling on a commodity PC as well as for fetching over the network, still it has enough information for statistical analysis.

Figure 2 is a sample output of *tcpdstat* from the *trans-pacific* point on February 12, 2000. This 1-hour-long trace contains about 2 million packets, the number of unique address pairs is about 56K. HTTP is dominant in the trace, 70% of the total packets and 62% of the total bytes.

Among the collected traces, some data sets contain traces of DoS attacks such as *portscan* and *smurf*. These traces could be useful for developing tools to detect such attacks.

The *6bone* point has been added in January 2000. The traffic volume of the *6bone* point is still low; the average rate is around 100Kbps and the majority of traffic is BGP and ICMPv6. However, we expect IPv6 traffic will increase in a few years as major router and OS vendors have started shipping IPv6 support in their base systems. Our intention is to record the evolution of IPv6 traffic in a long term.

Figure 3 is the output of *tcpdstat* from the *6Bone* point on the same day. This 3.5-hour long trace contains about 200K packets, the number of unique address pairs is about 270.

We expect that IPv6 traces will be useful for developing tools to support IPv6 since IPv6 traffic traces, especially on a backbone link, are not widely available.

Although we started collecting traces and made them available, we have not studied the traces thoroughly. Rather, one of the purposes of our open traffic repository is to leave analysis to those who are interested in doing it.

If other organizations start building similar but possibly closed traffic repositories, it would be possible to share experiences and development of tools. Especially, there are demands to develop a counter measure against the ever-growing threat of DoS attacks.

7 Future Work

Our focus at this moment is long-term data collection. So far, we have set sampling points only on relatively slow connections. Data collection from faster links is an obvious direction, but we have limited storage capacity and network capacity.

As for high-performance packet capturing, we can benefit from advanced research such as OC3MON [ACTW96]. OC3MON uses a DOS-based capturing tool to monopolize CPU, and takes advantage of the processor on the ATM card for offloading.

However, today's commodity PC is already quite powerful: Gigabit Ethernet is about 125MB/sec. The bus bandwidth of 32bit PCI at 33MHz is 132MB/sec, and 64bit PCI at 66MHz is 528MB/sec. The disk interface is getting faster as well. Ultra160 SCSI provides 160MB/sec. A single high-end disk has sustained rate of about 30MB/sec but disks can be used in parallel so that 4 disks provide about 120MB/sec. CPU power itself seems to be catching up.

There are also issues to run *tcpdump* on non-realtime UNIX; preemption could affect reliable data collection, resource contention and kernel-user data copy could af-

fect performance, network cards and drivers are not designed to obtain precise timestamp. Still, if the system is correctly tuned, a commodity PC seems to be capable of capturing packets even at a gigabit network.

8 Conclusion

We have presented the WIDE traffic repository, an ongoing effort to create archives of *tcpdump* files collected at several points within the WIDE backbone. Our attempt is a challenge to the legitimacy of concerns about revealing detailed traces for privacy and security reasons. We hope our repository will be useful for traffic analysis and for development of tools.

9 Availability

The WIDE traffic repository is at <http://tracer.csl.sony.co.jp/mawi/>. The traffic traces along with the tools we use can be downloaded from there. The traffic traces can be used only for research purposes. Actions that trespass upon users' privacy are prohibited.

References

- [ACTW96] J. Apisdorf, K. Claffy, K. Thompson, and R. Wilder. Oc3mon: Flexible, affordable, high performance statistics collection. In *Proceedings of LISA X*, pages 97–112, Chicago, IL, September 1996.
- [Bra88] R. T. Braden. A pseudo-machine for packet monitoring and statistics. In *Proceedings of SIGCOMM '88 Symposium*, pages 200–209, Stanford, California, August 1988.
- [CAI99] Coralreef. <http://www.caida.org/Tools/CoralReef/>, 1999.
- [Cis98] Cisco NetFlow. <http://www.cisco.com/warp/public/732/netflow/>, 1998.
- [DMPS95] Peter Danzig, Jeff Mogul, Vern Paxson, and Mike Schwartz. The Internet Traffic Archive. <http://ita.ee.lbl.gov/>, 1995.
- [JLM89] Van Jacobson, Craig Leres, and Steve McCanne. *tcpdump*. <ftp://ftp.ee.lbl.gov/>, 1989.
- [JLM94] Van Jacobson, Craig Leres, and Steve McCanne. *libpcap*. <ftp://ftp.ee.lbl.gov/>, 1994.
- [KMKA99] Akira KATO, Jun MURAI, Satoshi KATSUNO, and Tohru ASAMI. An internet traffic data repository: The architecture and the design policy. In *Proceedings of SOSP*, San Jose, CA, June 1999.
- [McR99] Daniel McRobb. *cflowd*. <http://www.caida.org/Tools/Cflowd/>, 1999.
- [MJ93] Steven McCanne and Van Jacobson. A BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of USENIX Winter Conference*, pages 259–269, San Diego, California, January 1993. Usenix.
- [MRA87] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of SOSP*, pages 39–51, Austin, TX, November 1987.
- [Oet96] Tobias Oetiker. MRTG: Multi Router Traffic Grapher. <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html>, 1996.
- [Oet99] Tobias Oetiker. RRDtool. <http://www.caida.org/tools/utilities/rrdtool/>, 1999.
- [tcp99] *tcpdump.org*. <http://www.tcpdump.org/>, 1999.
- [WID99] WIDE IPv6 Internet Exchange Point in Tokyo. <http://www.wide.ad.jp/nspixp6/>, 1999.
- [Ylo96] Tatu Ylonen. Thoughts on how to mount an attack on *tcpdpriv*'s “-a50” option... included with the *tcpdpriv* source distribution, 1996.

DumpFile: 200002121359.dump
 FileSize: 140.35MB
 Id: 200002121359
 StartTime: Sat Feb 12 13:59:00 2000
 EndTime: Sat Feb 12 15:06:29 2000
 TotalTime: 4048.47 seconds
 TotalCapSize: 108.37MB CapLen: 76 bytes
 # of packets: 2095754 (449.89MB)
 AvgRate: 932.21Kbps stddev:312.68K

Packet Size Histogram (including MAC headers)

```

[ 32- 63]: 1315693
[ 64- 127]: 258761
[ 128- 255]: 121532
[ 256- 511]: 113037
[ 512- 1023]: 137300
[ 1024- 2047]: 149431
  
```

IP flow (unique src/dst pair) Information

of flows: 56157 (avg. 37.32 pkts/flow)
 Top 10 big flow size (bytes/total in %):
 8.0% 6.0% 4.8% 2.8% 2.6% 2.0% 1.4% 1.2% 0.8% 0.7%

Protocol Breakdown

protocol	packets	bytes	bytes/pkt
total	2095754 (100.00%)	471744043 (100.00%)	225.10
ip	2095736 (100.00%)	471743089 (100.00%)	225.10
tcp	1768533 (84.39%)	400258906 (84.85%)	226.32
http	1474686 (70.37%)	292981631 (62.11%)	198.67
squid	42778 (2.04%)	36118457 (7.66%)	844.32
smtp	74280 (3.54%)	29130167 (6.17%)	392.17
nntp	1270 (0.06%)	101659 (0.02%)	80.05
ftp	23779 (1.13%)	7180413 (1.52%)	301.96
pop3	5601 (0.27%)	2537763 (0.54%)	453.09
telnet	995 (0.05%)	88678 (0.02%)	89.12
ssh	1950 (0.09%)	230243 (0.05%)	118.07
dns	1169 (0.06%)	94179 (0.02%)	80.56
bgp	6090 (0.29%)	419164 (0.09%)	68.83
other	135935 (6.49%)	31376552 (6.65%)	230.82
udp	264967 (12.64%)	62915121 (13.34%)	237.45
dns	187377 (8.94%)	29884111 (6.33%)	159.49
rip	135 (0.01%)	8910 (0.00%)	66.00
other	77455 (3.70%)	33022100 (7.00%)	426.34
icmp	51228 (2.44%)	5609707 (1.19%)	109.50
igmp	801 (0.04%)	48060 (0.01%)	60.00
ospf	8909 (0.43%)	2283318 (0.48%)	256.29
ipip	3 (0.00%)	246 (0.00%)	82.00
ip6	1295 (0.06%)	627731 (0.13%)	484.73
frag	112 (0.01%)	157111 (0.03%)	1402.78

tcpdump file: 200002121359.dump.gz (45.94 MB)

Figure 2: sample output of tcpdstat at trans-pacific

```

DumpFile: 200002120900.dump
FileSize: 17.64MB
Id: 200002120900
StartTime: Sat Feb 12 09:00:00 2000
EndTime: Sat Feb 12 12:33:45 2000
TotalTime: 12825.20 seconds
TotalCapSize: 14.69MB CapLen: 94 bytes
# of packets: 193424 (48.84MB)
AvgRate: 40.31Kbps stddev:63.46K

```

```

Packet Size Histogram (including MAC headers)
[ 64- 127]:    100654
[ 128- 255]:    66924
[ 256- 511]:    1179
[ 512- 1023]:   2322
[ 1024- 2047]:  22345

```

```

IP flow (unique src/dst pair) Information
# of flows: 270 (avg. 716.39 pkts/flow)
Top 10 big flow size (bytes/total in %):
53.9% 4.0% 3.8% 3.7% 3.6% 3.6% 3.1% 2.9% 2.9% 2.9%

```

Protocol Breakdown	packets	bytes	bytes/pkt
protocol			
total	193424 (100.00%)	51210692 (100.00%)	264.76
ip6	193424 (100.00%)	51210692 (100.00%)	264.76
tcp6	184430 (95.35%)	49453242 (96.57%)	268.14
smtp	402 (0.21%)	54893 (0.11%)	136.55
ftp	51229 (26.49%)	29569720 (57.74%)	577.21
ssh	53 (0.03%)	6820 (0.01%)	128.68
bgp	132476 (68.49%)	19798783 (38.66%)	149.45
other	270 (0.14%)	23026 (0.04%)	85.28
udp6	469 (0.24%)	36610 (0.07%)	78.06
other	469 (0.24%)	36610 (0.07%)	78.06
icmp6	7346 (3.80%)	1489628 (2.91%)	202.78
ip4	1179 (0.61%)	231212 (0.45%)	196.11

```

tcpdump file: 200002120900.dump.gz (2.99 MB)

```

Figure 3: sample output of tcpdstat at 6bone

JEmacs: The Java/Scheme-based Emacs

Per Bothner

<per@bothner.com>

Abstract

JEmacs is a re-implementation of the Emacs programmable text editor. It is written in Java, and uses the Swing GUI toolkit. Emacs is based on the extension language Emacs Lisp (Elisp), which is a dynamically-scoped member of the Lisp family. JEmacs supports Elisp, as well as the use of Scheme, a more modern statically-scoped Lisp dialect. Both languages get compiled to Java bytecodes, either in advance or on-the-fly, using the Kawa compilation framework.

1 Introduction

Emacs [7] [5] (in various versions) is a popular programmer's text editor. Emacs is programmable using "Emacs Lisp" (Elisp) [6] and many powerful packages are written in Elisp. The Free Software Foundation has a goal to replace Elisp with Scheme while also providing a translator to convert old Elisp files to Scheme. One reason is that Elisp is an ad-hoc, non-standard Lisp variant not used anywhere else, and not consistent with modern programming-language ideas. Another reason is that Guile, the primary GNU dialect of Scheme, is intended to be the standard extension language for GNU software, and so it makes sense for Emacs (the main GNU application with extensive use of a scripting language) to follow suit.

My opinion is that Guile is not the best Scheme implementation to use for Emacs. I happen to be biased, as I am the author of Kawa, a Java-based Scheme implementation. I also think that just replacing the extension language may not go far enough, and perhaps it is time to also replace the low-level code written in C. (One of the XEmacs maintainers told me he would really like to replace the re-display engine of XEmacs.)

Therefore, I have been working on a "next-generation" Emacs, based on Java and Kawa. The design includes:

- An implementation of the Elisp (core) syntax and language, such as functions used for creating lists and strings, defining functions, and macros.
- A set of Java classes based on the Swing GUI api that implement the Emacs "types", such as Buffer, Keymap, Window, Marker.
- A set of Scheme bindings to the Java methods. These are "similar to" and have the same names as standard Emacs Lisp functions, but are written in Scheme and intended to be called from Scheme.
- The equivalent Elisp functions: Implementations of the high-level Emacs functions as Elisp functions, so existing Elisp applications can (mostly) run without change.

The totality of these features is what I call "JEmacs". Below is a screenshot showing some of what has already been implemented. It includes a frame with a menubar, split into multiple windows, each with a mode line. The top window is a "Scheme interaction window", where Scheme expressions can be typed, and the result displayed. (Notice the user input is automatically boldfaced.) The user has just typed `C-x C-f`, which is bound to the function `find-file`. When `find-file` is called interactively or with no arguments, the `read-dialog` is called, which pops up the simple dialog window we see.

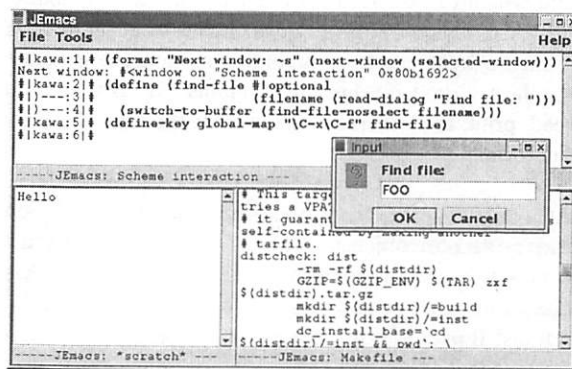


Figure 1: JEmacs in action

- An implementation of the Scheme language.

2 Motivation

This is a major, perhaps foolhardy, undertaking. Here are some reasons why it might make sense; I expand on these later.

- Swing is a modern GUI toolkit with good support for major Emacs concepts.
- Building on a Java run-time means we benefit from the work being done to run Java bytecodes fast.
- Java is multi-threaded.
- Kawa is a modern object-oriented Scheme, while Emacs is based on rather old design ideas.
- Java is based on Unicode and has good internationalization support.
- Java has lots of neat packages we can use.
- It would be useful to have Scheme (and Elisp) scripting for Swing applications.
- It is a good way to learn Swing!

3 Kawa

Kawa (see [2], [3]) is an implementation of the Scheme programming language written in Java. I have been developing Kawa since 1996. Unlike other such implementations, Kawa compiles Scheme into Java bytecodes, with non-trivial optimizations. It also provides almost all the other features you expect from a production Scheme system (including `eval` and `load`) and convenient interaction between Scheme and Java.

Kawa is also a framework used to implement Scheme, and which can be used for other languages. The package `gnu.bytecode` provides classes to generate Java `.class` bytecode files, including methods to generate the Java virtual machine instructions. It also lets you read, print, and otherwise manipulate the Java `.class` file format.

At a higher level, the `gnu.expr` package works on `Expression` objects. This is basically an abstract syntax tree (AST) representation, and the package has classes to generate and optimize expressions and declarations. It uses the `gnu.bytecode` package to generate bytecodes from the `Expression` representation.

The Kawa framework was originally used to compile Scheme. However, I wrote the beginnings of an EcmaScript (JavaScript) implementation, and others are using Kawa to compile other languages. For the JEmacs project, the framework is being used to compile Emacs Lisp to Java bytecodes, replacing the Emacs bytecode compiler: Instead of `.elc` files loaded into the Emacs bytecode interpreter, we use Java bytecode (`.class`) files loaded into a Java Virtual Machine.

4 Performance

A primary advantage of JEmacs is that Kawa is potentially much faster than either Elisp or Guile. Using an optimizing compiler that compiles to bytecode is certainly going to be faster than Guile or Emacs's simple interpreter. The Emacs bytecode-compiler uses the same idea, and produces a bytecode format that is more suitable to Emacs than Java bytecodes. However, there are many projects and companies working very hard on running Java bytecodes fast. The common approach is to use a "Just-in-Time compiler" (JIT), which dynamically compiles a bytecode method into native code inside the runtime. Another approach is to use a traditional "ahead-of-time" compiler (such as the Gcc-based Gcj [1]). It thus seems plausible (though unproven) that JEmacs can achieve better performance than Emacs.

5 The Swing Toolkit

Sun introduced Swing [4] in 1998 as the "next-generation" GUI toolkit for Java. Swing has a lot of functionality and many useful features. It builds on the earlier AWT (Abstract Windowing Toolkit). Of particular interest is that the text support in Swing is both very powerful, and also seems to be inspired by Emacs ideas. Swing has new "widgets" based on separating the "model" (data) and "view+control" (look+feel). For example, Swing distinguishes between a `Document` versus a `JTextComponent` that displays the `Document`, which is essentially the same as the Emacs buffer versus window distinction.

Swing also has a `Keymap` class similar to that of Emacs, and a `Position` that is like an Emacs marker. Unfortunately, neither of these are quite right for Emacs, but it was not difficult to create new classes that implement the Swing interfaces.

Swing has some other nice features, such as "pluggable-

look-and-feel” (themeability), a number of flexible “widgets”, and support for “structured” documents (i.e. XML/HTML structure in a buffer).

One problem with Swing is that while it is portable and freely redistributable, it does not have a free license, and there are no free re-implementations so far. (A related issue is that the documentation of Swing is very poor.) I’m hoping that by the time JEmacs becomes useable a free re-implementation of (the needed subset of) Swing will be available, and perhaps JEmacs will encourage this to happen. If not, we may re-write JEmacs so it can be built on top of some other free library (such as Gtk/Gnome or Qt).

Another possible problem with Swing is performance. Swing has the reputation for being slow. The first Emacs package ported to JEmacs (Towers-of-Hanoi animation) does run slower than under XEmacs. The cause of this is not clear, but it is quite possible it is due to Swing overheads. Perhaps in practice this may not matter, but it is certainly a possibility we may replace the use of Swing with a faster toolkit. More likely is replacing some of the implementation classes by alternative implementations; Swing is very flexible in this respect, because the API is defined in terms of abstract interfaces, rather than specific classes. (Some of these new implementation classes may also be useful in implementing a free Swing replacement.)

6 Multi-threading

One problem with traditional Emacs is that it is single-threaded. If you start some non-trivial operation (such as getting new mail), your Emacs session will be frozen until the operation completes. Java is designed to be multi-threaded, so it is in theory straightforward to create a multi-threaded Emacs.

One complication is that the Emacs Lisp execution model is inherently single threaded, since any Elisp function can change the current buffer or window to another buffer or window, while in the middle of the function. This means we cannot associate a thread with each buffer or with each window.

A solution to this problem is to use “buffer groups”, that is a group of related buffers and their windows which run in the same thread. Typically, there would be one buffer group for each Elisp “application”. By default, when an Elisp function creates a new buffer, it is put in the same buffer group as the the current buffer. However, an Elisp

function such as `find-file` can create a new buffer group when it creates a new buffer.

Another problem is that Swing is single-threaded. Only one thread (the event thread) can safely modify a buffer that is visible in a window. In the current JEmacs implementation all interactive commands are run by the event thread. Thus effectively, all of JEmacs is running inside the event thread. A solution is for long-running commands to use a “worker” thread. When the worker thread is finished, it lets the event thread know it is done, which can then update the buffers and display.

7 Java classes for Emacs

The following Java classes implement what we might call the Emacs data “model”.

- **Buffer:** An Emacs buffer. Contains a `Swing StyledDocument` object that manages the actual text (and styles). Contains a `BufferKeymap`, which manages the actions executed for different keystrokes.
- **BufferContent:** The actual characters of the `Buffer`. Implements the `Swing Content` interface. This class is needed because standard Swing does not support the `Marker` semantics we need.
- **Marker:** A position in a buffer that gets adjusted as needed. Similar to the `Swing Position` class, but also knows the `Buffer` it points to.

The following Java classes implement what we might call the Emacs “view+controller”.

- **BufferKeymap:** A data structure in one-to-one association with a `Buffer`. It implements the `Swing Keymap` interface, and manages the primitive `Keymaps`, to give the correct Emacs functionality.
- **Window:** Extends the `Swing JTextPane` class. Includes an associated `Modeline`, and a scrollbar.
- **Frame:** A top-level window. A `Frame` contains a nested hierarchy of `Windows`, sub-divided using Swing’s `JSplitPane`.

10.1 Syntax

While both Scheme and Elisp share the fully parenthesized prefix notation common to the Lisp family, there are some differences in the syntax of literals and identifiers. For example, the character 'a' is written `#\a` in Scheme, and `?a` in Elisp. The part of a Lisp system that converts a stream of characters to a value (usually a linked list) is traditionally called the "reader". We needed to write an Elisp reader to go along with the Scheme reader, and make sure the right one is invoked. This is fairly straightforward, and (except for some obscure features) done.

Once the reader has converted an input line or file to a list, the list needs to be converted into Kawa's internal Expression (abstract syntax tree) representation. This is handled similarly for Elisp and Scheme. However, Elisp has some new syntax forms (such as `defun`, `save-excursion`) and some forms that are different than in Scheme (such as `lambda`). For that we need to write new syntax transformers. This is almost done.

10.2 Symbols

The symbol data type in Scheme is very simple: It is an immutable atomic string; you can create a symbol from a (mutable) string, and you can convert the symbol back to a string (for example for printing). Whenever you convert a string to a symbol, you will always get the same identical symbol, as long as the strings have the same characters. This process is called *interning* and is implemented using a global hash-table. Symbols are used for multiple purposes, but the most important one is that identifiers in a Scheme program are represented using symbols.

Java has a similar datatype, the class `String`, which is used all over the place in Java. Java has a method, called `intern`, which returns an interned version of the `String`. This functionality is exactly what is needed for Scheme, so Kawa uses `String` for Scheme symbols. This has the side benefit of increasing interoperability between Scheme and Java.

On the other hand, an Elisp symbol has extra properties: value and function bindings, and a property list. The traditional implementation is that a symbol value is a pointer to a structure containing the necessary fields. This makes extracting the value and function bindings cheap, but it requires extra space in all symbols. JEmacs uses an alternative implementation: an Elisp symbol

value is a reference to a `String` instance, just as in Kawa. To get the value or function binding of a symbol, you lookup the symbol in the current `Environment`. This yields a `Binding`, which contains the needed fields. For symbols that are used as identifiers in a function, the compiler generates code to get the `Binding` when the function is loaded. Since we don't have to do a hashtable lookup when the function is executed, symbol lookup is about as fast as in the traditional implementation.

10.3 Nil - the empty list

In Elisp, the empty list and the symbol `'nil` are the same object, but in Scheme they are different. There are various ways to deal with the problem, none particularly elegant. In Kawa, lists inherit from the abstract `Sequence` class. I feel it is important that the empty list also be a `Sequence`, even for Elisp, and it is important to be able to pass lists between Scheme and Elisp code. I decided that `'nil` would have to be a special case: While the Elisp symbol `t` is represented using the `String "t"`, the symbol `nil` is represented by the special `LList` value `LList.Empty` that Kawa uses for empty lists. Thus the predicate `(symbolp x)` is implemented as `(x == LList.Empty || x instanceof java.lang.String)`.

10.4 Standard Elisp Functions

Elisp has many builtin functions and macros which are different from Scheme. There is no fundamental difficulty with this; just a lot of porting/conversion work. Many of the basic editing functions are already implemented, but many (such as those involving searching) are not.

11 Variables

Variable lookup is different in Scheme and Elisp in two main ways: Elisp uses dynamic scoping, while Scheme uses lexical scoping; and Elisp has different namespaces for function names and variable names, while Scheme has a single namespace for both. The latter is an easy matter of the compiler emitting the code to look for the name in the correct namespace. Handling dynamic scoping is done using Kawa's support for the `fluid-name` form, which provides dynamic binding using a very flexible name-binding mechanism.

11.1 Constrained Variables

In Kawa, each global variable is a Binding object. A Binding has an optional name, a value field, and a Constraint. The constraint contains the actual methods that get/set the value of the Binding. For example, setting `b` to `x` does `b.constraint.set(b, x)`.

The default action for `get` retrieves the Binding value field. Different sub-classes of Constraint have different implementations of `get` and `set`. If there is a thread-local dynamic or buffer-local binding, we just put the appropriate constraint in the binding.

This framework can handle indirection, unbound variables (`get` throws an exception), and constraint propagation. Changing a value can trigger arbitrary checks or notification messages.

12 Modes

In Emacs, a mode is a set of keybindings, functions, and variables local to a buffer. There is no object corresponding to a “mode”, but there is a set of conventions to follow. In an object-oriented environment it seems better to define a separate *mode class* for each mode. Each buffer that has a mode enabled should have a corresponding *mode instance*.

Each buffer has a linked list of mode instances, one for each major/minor mode that is active for the buffer. Mode functions are compiled to virtual methods of the mode object. Instead of a buffer-local variable, use a field of the mode object. This provides fast access to variables in compiled code, without run-time symbol lookup. A derived mode can use mode class inheritance.

As an example, the abstract class `ProcessMode` inherits from the generic `Mode` class. A `ProcessMode` represents some kind of *process* which (usually) generates output that gets inserted into the buffer. Partial implementations exist of the sub-classes `InfProcessMode`, which displays the output of an external (possibly-interactive) program, and `TelnetMode`. Both of these provide minimal terminal emulation. A full terminal emulator would be desirable, though line wrapping is a complication. (Swing handles line wrapping on word boundaries, as expected by people used to word processor, but a normal terminal emulator wraps on character boundaries. The best solution is probably to write a custom `View` class.)

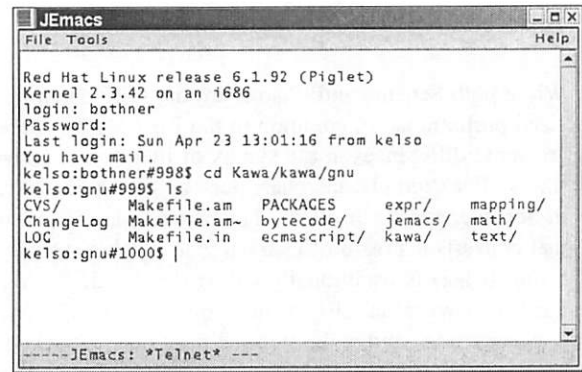


Figure 3: Telnet mode

The Mode framework has been used for modes written directly in Java. It is not yet clear how to write a mode using Scheme, nor whether legacy Emacs code can be automatically compiled into classes that inherit from `Mode`.

13 Unicode and Internationalization

The Java `char` is a 16-bit Unicode character. Internal strings and JEmacs buffers use these Unicode chars. On the other hand, external files consist of 8-bit bytes. So Java provides named encodings that map byte streams and character streams.

Java with Swing handles much of the work needed for complex text processing. For example, bi-directional text (as needed for Hebrew and Arabic) is taken care of. In the screenshot below, we have a file encoded in UTF-8. It contains a string of four Hebrew characters, stored in the buffer in logical (reading) order. When they displayed, they are shown right-to-left. Notice that if you make a text selection that includes both English and Hebrew text, the selected characters form a contiguous region in the memory, but because different segments are displayed in different order, in the window we see the selection as two non-contiguous pieces.

Also note that characters not present in the font are displayed as hollow rectangles. I.e. I need to install a more complete font!

Current releases of Emacs and XEmacs support some internationalization, using the Mule framework which supports text in many character sets. However, current releases of Mule do not yet support Unicode, which is where most of the world is heading. Mule is based on

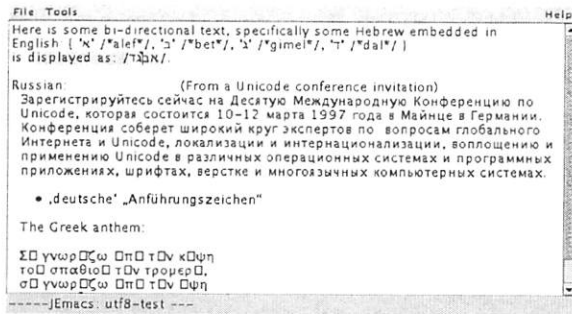


Figure 4: Multiple scripts

some rather dated design decisions. JEmacs will not support Mule; it won't need to.

14 Word Processing and XML

Documents will be increasingly represented using XML externally and DOM (Document Object Model) internally. Most of the programs and libraries for manipulating and formatting XML are written in Java. For example, the Apache group has some major Java-based XML projects.

If Emacs is to support word processing features, it should build on XML standards. It is easier to use third-party Java libraries if the Emacs core is Java-based.

The plan is for JEmacs to implement a class that implements the XEmacs “extent” API, and also implements the Swing Element interface, as well as the DOM Node interface.

JEmacs will use some form of “virtual document” that implements the Swing Document interface, but gets its content indirectly from other documents. The class `AbstractString` is an abstract class that generalizes strings, buffers, shared substrings, buffer regions, and general indirection. This is part of a design to support editable documents which are defined using transformations from other documents (rather like a database “view”).

15 Status and Conclusion

A “proof-of-concept” prototype is working, including partial Elisp implementation. There is still a lot of work before you want to use JEmacs for day-to-day general

editing.

Using the libraries of Java and Swing takes care of many problems.

JEmacs has a mailing list and a home page (<http://JEmacs.SourceForge.net/>).

JEmacs is currently distributed together with Kawa (<http://www.gnu.org/software/kawa/>).

References

- [1] Per Bothner. *A Gcc-based Java Implementation*. IEEE Compcon '97, 1997.
- [2] Per Bothner. *Kawa - Compiling Dynamic Languages to the Java VM*. Unix Annual Technical Conference, 1998.
- [3] Per Bothner. *Kawa: Compiling Scheme to Java*. Lisp Users Conference, 1998.
- [4] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, ISBN 0-201-43321-4, 1999.
- [5] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 1985.
- [6] Bil Lewis, Daniel LaLiberte, and GNU manual group. *GNU Emacs Lisp reference manual*. Free Software Foundation, 1990.
- [7] Richard Stallman. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*. In Text manipulation: Proceedings of the ACM SIGPLAN/SIGOA symposium (Portland, OR, June, 1981), June, 1981.

A New Rendering Model for X

Keith Packard

XFree86 Core Team, SuSE Inc.

keithp@suse.com

Abstract

X version 11 [SG92] was originally designed and implemented in 1987. In the intervening 13 years, there have been advancements in both applications and hardware, but the core of the X Window System has remained largely unchanged. The last major X server architecture changes were included in X11R4. The last wide-spread functional enhancement exported by the X server might well be the Shape extension [Pac89], designed (in the hot tub) at the 1989 Winter Usenix in San Diego.

The rise of inexpensive Unix desktop systems in the last couple of years has led to the development of new user-interface libraries, which are not well served by the existing X rendering model. A new 2D rendering model is being developed to serve this new community of applications. The problem space and proposed solutions are discussed.

1 Introduction

While a window system is more than a collection of rendering routines, the available rendering primitives constrain the capabilities of applications more than anything else. The X rendering model was developed to match the abilities of workstation hardware developed fifteen years ago and has significant limitations when applied to application development today.

As application development has advanced, the X protocol has devolved into little more than an image transport mechanism. Applications perform rendering in client-side buffers and transport the result to the screen. A shared memory mechanism for delivering images to the X server exists when the application is running on the same machine as the dis-

play, but performance suffers when attempting to run these applications over the network.

Many new graphics accelerators are providing acceleration for operations needed by new applications. Only by moving these operations into the X server can this acceleration be made accessible to X applications.

2 Origins of X Rendering

A combination of archaeology and history is needed to understand the current state of X rendering technology. Cast your mind back to 1987, and try to remember graphical workstations of that era. A 1 MIPS machine was the state of the art and one was lucky to have color on the desktop. Color, of course, was 8 bits with a palette. Those hotheads over at SGI were making noises about true color hardware, but for most that was not even a dream. Hardware acceleration was available, but frequently no faster than software, and a huge pain to code for.

The state of the art in 2D rendering was PostScript [Ado85]. The definition of objects by precise mathematical formulae was compellingly beautiful to engineers. PostScript provided sophisticated font technology embedded inside the printers of the era, but left the desktop with only bitmap versions of the same fonts.

Into this stepped a group of networking protocol and hardware hackers intent on updating their latest offering, the X Window System. Not a single one of them had even been introduced to a computational geometer, nor did they have the resources of the modern internet to help with the design. Of course a constant refrain was to get the darn thing finished and out the door. Digital, who was funding the sample implementation, had product schedules to

meet. Meanwhile, back at MIT, Project Athena was deploying more and more X10 boxes.

So they picked up the PostScript “Red Book” and started writing a specification. Of course their new window system was extensible; with any luck, limitations in the original design would be masked by clever add-ons in the future. What they failed to realize was that the Red Book inadequately described the actual implementation of some primitives. The developers also lacked foresight about how difficult it would be to create consensus around future rendering standards.

One big limitation of PostScript in that era was in image manipulation. Printers were black-and-white, so PostScript didn’t need any complex image compositing operators. Besides, X was an interactive protocol: alpha blending a full-screen image looked like slugs racing down the monitor.

And then there were lumpy lines. The Red Book describes a beautifully pure line stroking algorithm: a circular pen is dragged along the path and illuminates pixels within the circle. Too bad that the results look ugly—the apparent width of the line varies along the length of the line. Lacking understanding of the problem, Adobe kludged around it. John Hobby had recently solved the problem [Hob85], but his solution had not yet been published outside of Stanford and was not discovered by the X community for several years.

Instead of providing PostScript paths, X provided only straight lines and axis-aligned ellipses. Why axis-aligned? Because there was a rumor that the rendering algorithm for thin non-axis aligned ellipses was patented and there was agreement that X should be free of patented technologies. This rumor was unfounded; the algorithm (published many years ago [Pit67]) was unencumbered.

At one meeting, members of the X11 team looked around the table and discovered that not one of them had any clue about splines. Instead of doing something wrong, they left them out. Sub-pixel positioning was deemed an extravagant use of network bandwidth, since it would double the payload of each rendering primitive by requiring the use of 32 bits for each coordinate instead of 16.

The expectation was that these issues could be left for future development in the form of an extension. However, the usage of X expanded and compatibil-

ity between X servers was deemed a market necessity. Creating an extension that existed in only some X servers would create application interoperability problems. Thus the rendering model has stagnated.

2.1 Problems with the Core Protocol

Even ignoring new rendering techniques, the core protocol rendering architecture has some fundamental problems:

Lack of a stenciling operator

X10 provided a stenciling operator for solid fills, even this operator is missing from X11. A stencil can be emulated using a ClipMask, but the sample implementation of ClipMasks is inefficient, making this impractical.

Stenciling can be used to accelerate missing rendering primitives, the application generates the appropriate shape in a monochrome bitmap and uses that to stencil the result to the screen. The implementors of the sample server knew this and included a stenciling operator inside the server for use by higher level primitives.

Separation of lines and arcs

As useless as axis-aligned arcs are, they are made even less useful by being separated from lines. This means there is no way to join a sequence of lines and arcs together. As a special case, zero width/height arcs are defined to be equivalent to lines, making it possible to render an axis-aligned rounded rectangle.

No vertical escapement for text

This is all that is needed to render Asian text and to allow for rotated fonts.

2.2 Features of the Core Protocol

In building a new rendering system, it would be unwise to ignore the best parts of the existing system:

Precise pixelization

Each X operator, with the exception of thin primitives, has exactly specified pixelization requirements. This not only allows for reproducible rendering across X server implementations, but probably more importantly allows for

automated testing of the rendering code. The rules themselves may be broken, but their existence is of vital importance.

Pixel values, not colors

Providing an underlying pixel value basis for the rendering system allows for the implementation of a color-based system in user space. The reverse is not true. Additionally, the only way to make boolean pixel operators usable by applications is to expose the pixel values.

Allow all rendering permutations

X allows applications to render stippled text using a variety of raster-ops (such as XOR). Such combinations work with all primitives other than ImageText. This makes it possible to dither everything on the screen in a consistent manner or to apply a reversible XOR raster-op.

3 Reasons for a New Model

The strongest argument for building a new rendering model is in evidence on almost every Linux machine these days. The combination of KDE, Gnome, and Enlightenment demonstrate that the world of 2D graphics is rapidly leaving the X Window System behind. These applications use sophisticated rendering primitives like outlined text and cubic splines. They improve image quality with anti-aliasing and blend images together with alpha compositing.

It is no longer a question of what kind of rendering will be done. The question now is where that rendering should happen. Applications will advance, and X must either keep up or get out of the way. One thing working in favor of an extension today is that many new applications are being written using a higher-level rendering model provided by a toolkit. Providing new X server functionality that matches the rendering model in the toolkit allows for a gradual adoption of the extension as the toolkits are modified: the toolkits can accelerate operations using the extension when available and still fall back to client-side rendering for older X servers.

4 Components of a New Rendering System

The current generation of 2D applications are similar in their demands on the rendering system. By analyzing existing usages and choosing primitives with care, a reasonably consistent system can be built which will be useful for many applications. The existence of applications with well-understood requirements provides an opportunity lacking in the initial protocol design.

4.1 Alpha Compositing

Alpha compositing is the blending together of images with a per-pixel (α) value controlling an arithmetic combination of the colors. There are many reasonable functions for this operator. The most common is a translucency operation, in which the colors are combined as $v = \alpha v_1 + (1 - \alpha)v_2$. As images are composited with this operator, they appear as translucent overlays on the original image.

Alpha compositing is also useful in approximating anti-aliasing. A suitable function and constraints on both the structure and order of the rendering primitives can yield satisfactory results.

3D applications make significant use of alpha compositing, so graphics hardware now commonly supports some of the most popular alpha compositing functions of OpenGL. Giving applications access to hardware compositing will provide dramatic performance improvements.

There are many different ways of presenting image data along with alpha channel information. At 32 bits per pixel, the alpha channel is frequently delivered in the unused upper byte. For 16 bit images sometimes the alpha channel is embedded as one of four 4-bit components and sometimes the alpha channel is in a separate 8-bit image.

For applications to be able to take maximal advantage of the available acceleration, the characteristics of the hardware must be exposed to the application. This significantly complicates the toolkit, which must match rendering requests with available resources.¹

¹Better architectural ideas are welcome.

Alpha compositing is easy to describe in a TrueColor environment, but more problematic in PseudoColor where there is no linear relation between pixel and color. Fortunately, most modern machines are able to display in TrueColor, making it tempting to provide this functionality only in that case.

4.2 Anti-Aliasing

Anti-aliasing is the application of signal processing in rasterization. It reduces the high-frequency quantization noise generated by imprecisely positioned object edges. Conceptually, anti-aliasing is performed by oversampling the image and resampling at the screen resolution.

A direct approach would create an oversampled version of the image in memory, and resample the completed image either to the frame buffer or (ideally) as it is delivered to the screen. The prospect of multiplying the amount of video memory by some large amount and reducing rendering performance by a similar amount have led to a search for inexpensive incremental approximations.

When displaying a single convex primitive, the simple alpha compositing operator described above can be used to accurately approximate anti-aliasing. By generating an alpha channel containing the output of the resampling filter, the primitive can be composited onto the screen. However, when more than one primitive is involved the task becomes more difficult, as the alignment of the edges of each primitive is lost in the compositing operation.

OpenGL contains a set of more complicated alpha operations, which ameliorate the errors in this approximation when used properly. A reasonable subset of these operations will be included in the new system.

As mentioned above, the alpha channel is filled with the output of the resampling filter. Most existing anti-aliasing systems simply compute the amount of the pixel covered by the object and use that as the alpha value; for the edges of a polygon, the system has a measure of that value computed as it walks the edge. A more sophisticated anti-aliasing system uses the output of a 2D filter to fill the alpha channel. This filter can even take into account the response characteristics of the electron beam displaying the image: systems built with such techniques

work quite well.

Given that this alpha blending technique is only approximate and that sophisticated techniques are likely to be a performance problem in the near term, only the simple coverage model is currently planned. Provisions will be made for adding new anti-aliasing mechanisms in the future.

4.3 Coordinate System

The current rendering system uses a 16-bit integer coordinate space, which is fine for describing rectangles but imprecise when drawing text lines and polygons. Sub-pixel positioning is essential when compositing polygons into larger shapes, to avoid visible discontinuities along edges.

Sub-pixel positioning allows applications to more precisely position objects on the screen. To render an object using the core protocol, the coordinates must be rounded to the nearest pixel boundary. This mispositions the object by as much as 1/2 pixel. While this may not seem serious, the cumulative visual effect of many 1/2 pixel errors is quite noticeable. Scott Nelson describes this problem in more detail [Nel96], including an example showing the improvement offered by sub-pixel positions even in the absence of anti-aliasing.

One obvious coordinate representation is IEEE 32-bit floating point numbers. The 24 bit mantissa specified by IEEE would provide at least 8 bits of sub-pixel position within the 16-bit X coordinate space, and would be easy for applications to manage.

However, it is desirable for objects to be translationally invariant. As objects move to larger coordinates, IEEE floats will slowly drop bits of sub-pixel position information. This is especially important as windows move around the screen. While IEEE floats could probably be made to work by artificially limiting their precision for smaller values, using fixed-point numbers eliminates this problem entirely.

The next question is how many bits of fraction to use. Four is enough for most applications, but eight will suffice for all but the most particular uses. Applications which use larger coordinate spaces will still need to perform clipping operations during the

transformation to X coordinates, but with 8-bits of sub-pixel position, it should suffice for most to simply truncate objects at the boundary of the X coordinate space.

For these reasons, 32-bit fixed-point coordinates with 8 fractional bits will be used.

4.4 Rendering Primitives

One thing missing from the core protocol is a simple server primitive that could be used to render geometrical objects not defined by the protocol. Inside a PostScript interpreter, the primitive used is a horizontal trapezoid—that is, the top and bottom edges are horizontal. LibArt, the rendering library for the Gnome project, uses an equivalent primitive called sorted edge lists.

So, at a minimum, this new primitive will be included. Unlike the core polygon request, this request will be able to draw many trapezoids at a time.

A question remains as to whether PostScript-style paths should be included. Doing so would significantly reduce the wire traffic but would complicate the implementation. The paths would include lines, cubic splines and character elements.

Paths would be rasterized by cracking them into trapezoids as described above, using a settable error value to describe the polygonalization of curves. By making this rendering mechanism explicit, it would be possible to precisely specify pixelization of the path in relatively simple terms, and to exactly replicate this pixelization on the client side if necessary.

4.5 Text

The original X design was done before outline rasterizers were used to generate screen fonts. The only fonts available were bitmaps, and the idea of providing scaled versions of those for the screen lacked appeal.

The resulting design does not match the realities of outline fonts well at all. Even the XLFD specification (and its extensions to support scalable fonts, font subsetting, and glyph rotation) is difficult or impossible to use with outlined fonts.

One problem to be solved is in the naming and accessing of fonts. A simple mechanism could be added to provide more control over which font is selected, and to provide more than a simple string to identify fonts. Another issue is access to additional metrics about the font, such as pair kerning tables, glyph names, and more precise glyph metrics.

A requirement for modern applications is that the application and the X server share access to the raw outline data and metrics. This allows the application to augment the text rendering provided by the X server with fancier versions on the client side. An easy way to provide this is to extend the X Font Services Protocol [Ful94] to include this additional information.

Another issue with the core protocol is in accessing glyph metrics. The core protocol provides only the QueryFont request which retrieves metrics for all glyphs in a font at once. This allows the client to quickly compute the extents for any set of glyphs without consulting the server in the future. However, it also requires that the metrics for every glyph in the font be available when the request is made. For scalable fonts, this means that the entire font must be rasterized; for most scalable technologies, generating X metrics is a side effect of rasterizing glyphs.

Most applications issue a QueryFont for each font that they open, this means that in normal usage, the X server rasterizes every glyph in every font used by applications.

Additionally, the ListFontsWithInfo request returns bounding metrics for all glyphs in the font. Computing the bounding metrics requires the complete set of metrics for the font.

New font information requests are needed. A request to query the metrics for a list of glyphs along with a new font listing function which provides as much information about the font as can be gathered without rasterizing every glyph.

Better rendering primitives are required as well, allowing for rotation of glyphs and baselines, sub-pixel positioning, and anti-aliasing.

Direct support of glyph outlines may be addressed at some point. This is somewhat difficult given the multiplicity of outline font formats, the lack of high-quality Type1 rasterizers and the additional render-

ing infrastructure required.

5 Strategy

Building a new rendering system will take some time, and feedback during the process is essential to make it successful. To make this possible, the system will be developed in stages, with each stage building on the previous stages. Some enhancements will be available soon, while others wait both for resources to implement them and for consensus to be built supporting the particular design.

1. Alpha Compositing

Many applications need this today but are suffering with unaccelerated client-side implementations. This is an operation that graphics hardware can improve by a huge amount, forming the basis for anti-aliased graphics.

2. Trapezoids

Moving these primitives to the server will reduce the demands placed on the bus between the CPU and the graphics adapter.

3. Paths

Moving these into the server will reduce wire traffic, but not provide any dramatic performance improvements except in a networked environment.

5. Font Information

Reducing the work required to open and list fonts will improve the ability of the system to cope with the increasing availability of outline and 16-bit fonts.

4. Font Access

Improving the mechanisms by which the X server and application share access to the same fonts will allow for improvements in management and deployment of applications, especially in a complex networked environment.

5. Text Rendering

Adding the ability to display outline fonts with the option of anti-aliasing has been on the "wish list" for a long time.

Each of these systems will be implemented first in software, and then hardware acceleration will be

provided for some common graphics chips. Where possible, existing graphics systems can be used to avoid a duplication of effort. In particular, OpenGL will make this task easier for chips which have appropriate support in place.

6 Conclusion

The existing X rendering model was rushed to completion by people who understood their limitations and expected it to be quickly augmented with suitable extensions. No credible 2D graphics extensions have been developed in the intervening 13 years, but the world has recently changed. The advent of new toolkits that provide advanced rendering models abstracted from the core protocol opens a new opportunity to improve the X Window System. A new rendering model, designed to solve specific performance and network transparency issues of these new toolkits, has the promise of significantly increasing the power of the X desktop environment.

References

- [Ado85] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, 1985.
- [Ful94] Jim Fulton. The x font service protocol. X consortium standard, Network Computing Devices, Inc., 1994.
- [Hob85] John D. Hobby. *Digitized Brush Trajectories*. PhD thesis, Stanford University, 1985. Also *Stanford Report STAN-CS-85-1070*.
- [Nel96] Scott R. Nelson. Twelve characteristics of correct antialiased lines. *Journal of Graphics Tools*, 1(4):1-20, 1996.
- [Pac89] Keith Packard. X nonrectangular window shape extension protocol. X consortium standard, MIT X Consortium, 1989.
- [Pit67] M. L. V. Pitteway. Algorithm for drawing ellipses or hyperbolae with a digital plotter. *The Computer Journal*, 10(3):282-289, November 1967.
- [SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD

Chuck Silvers

The NetBSD Project

chuq@chuq.com, <http://www.netbsd.org/>

Abstract

This paper introduces UBC ("Unified Buffer Cache"), a design for unifying the filesystem and virtual memory caches of file data, thereby providing increased system performance. In this paper we discuss both the traditional BSD caching interfaces and new UBC interfaces, concentrating on the design decisions that were made as the design progressed. We also discuss the designs used by other operating systems to solve the same problems that UBC solves, with emphasis on the practical implications of the differences between these designs. This project is still in progress, and once completed will be part of a future release of NetBSD.

1 Introduction

Modern operating systems allow filesystem data to be accessed using two mechanisms: memory mapping, and I/O system calls such as `read()` and `write()`. In traditional UNIX-like operating systems, memory mapping requests are handled by the virtual memory subsystem while I/O calls are handled by the I/O subsystem. Traditionally these two subsystems were developed separately and were not tightly integrated. For example, in the NetBSD operating system[1], the VM subsystem ("UVM"[2]) and I/O subsystem each have their own data caching mechanisms that operate semi-independently of each other. This lack of integration leads to inefficient overall system performance and a lack of flexibility. To achieve good performance it is important for the virtual memory and I/O subsystems to be highly integrated. This integration is the function of UBC.

2 Background

In order to understand the improvements made in UBC, it is important to first understand how things work without UBC. First, some terms:

- "buffer cache":

A pool of memory allocated during system startup which is dedicated to caching filesystem data and is managed by special-purpose routines.

The memory is organized into "buffers," which are variable-sized chunks of file data that are mapped to kernel virtual addresses as long as they retain their identity.

- "page cache":

The portion of available system memory which is used for cached file data and is managed by the VM system.

The amount of memory used by the page cache can vary from nearly 0% to nearly 100% of the physical memory that isn't locked into some other use.

- "vnode":

The kernel abstraction which represents a file.

Most manipulation of vnodes and their associated data are performed via "VOPs" (short for "vnode operations").

The major interfaces for accessing file data are:

- `read()` and `write()`:

The `read()` system call reads data from disk into the kernel's cache if necessary, then copies data from the kernel's cached copy to the application's address space. The `write()` system call moves data the opposite direction, copying from the application's address space into the kernel's cache and eventually writing the data from the cache to disk. These interfaces can be implemented using either the buffer cache or the page cache to store the data in the kernel.

- `mmap()`:

The `mmap()` system call gives the application direct memory-mapped access to the kernel's page cache data. File data is read into the page cache lazily as processes attempt to access the mappings created with `mmap()` and generate page faults.

In NetBSD without UBC, `read()` and `write()` are implemented using the buffer cache. The `read()` system call reads file data into a buffer cache buffer and then copies it to the application. The `mmap()` system call, however, has to use the page cache to store its data since the buffer cache memory is not managed by the VM system and thus cannot be mapped into an application address space. Therefore the file data in the buffer cache is copied into page cache pages, which are then used to satisfy page faults on the application mappings. To write modified data in page cache pages back to disk, the new version is copied back to the buffer cache and from there is written to disk. Figure 1 shows the flow of data between the disk and the application with a traditional buffer cache.

This double-caching of data is a major source of inefficiency. Having two copies of file data means that twice as much memory is used, which reduces the amount of memory available for applications. Copying the data back and forth between the buffer cache and the page cache wastes CPU cycles, clobbers CPU caches and is generally bad for performance. Having two copies of the data also allows the possibility that the two copies will become inconsistent, which can lead to application problems which are difficult to debug.

The use of the buffer cache for large amounts of data is generally bad, since the static sizing of the buffer cache means that the buffer cache is often either too small (resulting in excessive cache misses), or too large (resulting in too little memory left for other uses).

The buffer cache also has the limitation that cached data must always be mapped into kernel virtual space, which puts an additional artificial limit on the amount of data which can be cached since modern hardware can easily have more RAM than kernel virtual address space.

To solve these problems, many operating systems have changed their usage of the page cache and the buffer cache. Each system has its own variation, so we will describe UBC first and then some other popular operating systems.

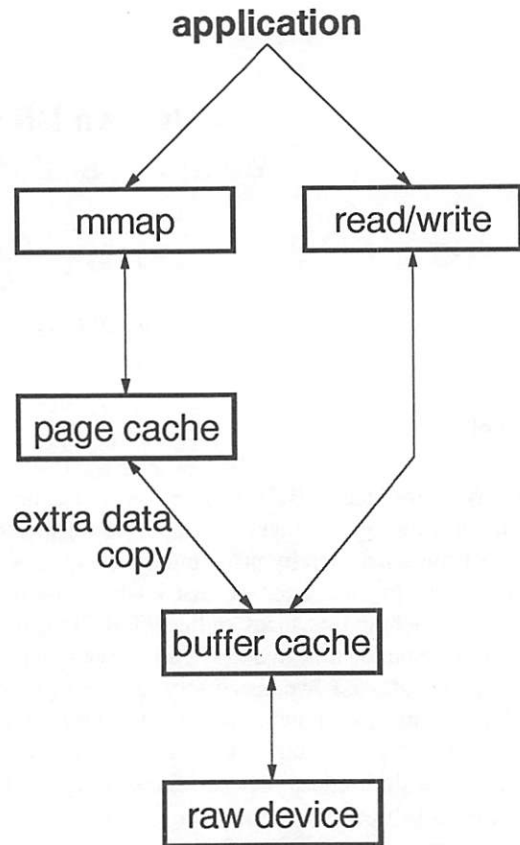


Figure 1: NetBSD before UBC.

3 So what is UBC anyway?

UBC is a new subsystem which solves the problems with the two-cache model. In the UBC model, we store file data in the page cache for both `read()/write()` and `mmap()` accesses. File data is read directly into the page cache without going through the buffer cache by creating two new VOPs which return page cache pages with the desired data, calling into the device driver to read the data from disk if necessary. Since page cache pages aren't always mapped, we created a new mechanism for providing temporary mappings of page cache pages, which is used by `read()` and `write()` while copying the file data to the application's address space. Figure 2 shows the changed data flow with UBC.

UBC introduces these new interfaces:

- `VOP_GETPAGES()`, `VOP_PUTPAGES()`

These new VOPs are provided by the filesystems to allow the VM system to request ranges of pages

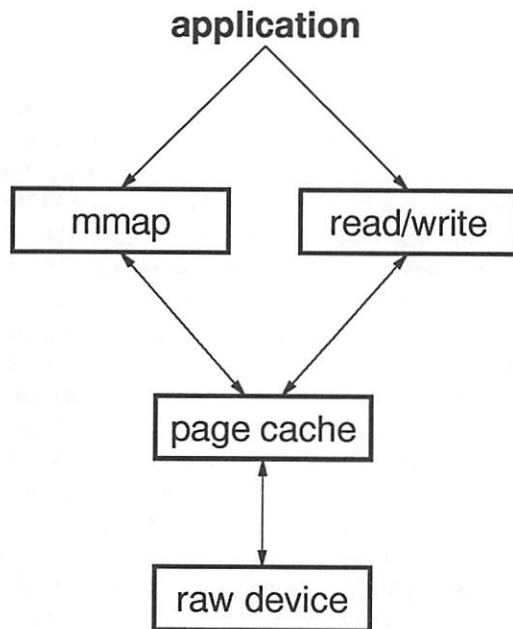


Figure 2: NetBSD with UBC.

to be read into memory from disk or written from memory back to disk. `VOP_GETPAGES()` must allocate pages from the VM system for data which is not already cached and then initiate device I/O operations to read all the disk blocks which contain the data for those pages. `VOP_PUTPAGES()` must initiate device I/Os to write dirty pages back to disk.

- `ubc_alloc()`, `ubc_release()`

These functions allocate and free temporary mappings of page cache file data. These are the page cache equivalents of the buffer cache functions `getblk()` and `brelse()`[3]. These temporary mappings are not wired, but they are cached to speed repeated access to the same file. The selection of which virtual addresses to use for these temporary mappings is important on hardware which has a virtually-addressed CPU data cache, so the addresses are carefully chosen to be correctly aligned with the preferred addresses for user file mappings, so that both kinds of mappings can be present at the same time without creating coherency problems in the CPU cache. It is still possible for applications to create unaligned file mappings, but if the application lets the operating system choose the mapping address then all mappings will always be aligned.

- `ubc_pager`

This is a UVM pager which handles page faults on

the mappings created by `ubc_alloc()`. (A UVM pager is an abstraction which embodies knowledge of page-fault resolution and other VM data management. See the UVM paper[2] for more information on pagers.) Since its only purpose is to handle those page faults, the only action performed by `ubc_pager` is to call the new `VOP_GETPAGES()` operation to get pages as needed to resolve the faults.

In addition to these new interfaces, several changes were made to the existing UVM design to fix problems which were glossed over in the original design.

Previously in UVM, `vnodes` and `uvm_objects` were not interchangeable, and in fact several fields were duplicated and maintained separately in each. These duplicate fields were combined. At this time there's still a bit of extra initialization the first time a `struct vnode` is used as a `struct uvm_object`, but that will be removed eventually.

Previously UVM only supported 32-bit offsets into `uvm_objects`, which meant that data could only be stored in the page cache for the first 4 GB of a file. This wasn't much of a problem before since the number of programs which wanted to access file offsets past 4 GB via `mmap()` was small, but now that `read()` and `write()` also use the page cache interfaces to access data, we had to support 64-bit `uvm_object` offsets in order to continue to allow any access to file offsets past 4 GB.

4 What do other operating systems do?

The problems addressed by UBC have been around for a long time, ever since memory-mapped access to files was first introduced in the late 1980's. Most UNIX-like operating systems have addressed this issue one way or another, but there are considerable differences in how they go about it.

The first operating system to address these problems was SunOS[4, 5], and UBC is largely modeled after this design. The main differences in the design of the SunOS cache and UBC result from the differences between the SunOS VM system and UVM. Since UVM's pager abstraction and SunOS's segment-driver abstraction are similar, this didn't change the design much at all.

When work on UBC first began over two years ago, the

other design that we examined was that of FreeBSD[6], which had also already dealt with this problem. The model in FreeBSD was to keep the same buffer cache interfaces to access file data, but to use page cache pages as the memory for a buffer's data rather than memory from a separate pool. The result is that the same physical page is accessed to retrieve a given range of the file regardless of whether the access is made via the buffer cache interface or the page cache interface. This had the advantage that filesystems did not need to be changed in order to take benefit from the changes. However, the glue to do the translation between the interfaces was just as complicated as the glue in the SunOS design and failed to address certain deadlock problems (such as an application calling `write()` with a buffer which was a memory mapping of the same file being written to), so we chose the SunOS approach over this one.

The approach taken by Linux[7] (as of kernel version 2.3.44, the latest version at the time this paper was written) is actually fairly similar to the SunOS design also. File data is stored only in the page cache. Temporary mappings of page cache pages to support `read()` and `write()` usually aren't needed since Linux usually maps all of physical memory into the kernel's virtual address space all the time. One interesting twist that Linux adds is that the device block numbers where a page is stored on disk are cached with the page in the form of a list of `buffer_head` structures. When a modified page is to be written back to disk, the I/O requests can be sent to the device driver right away, without needing to read any indirect blocks to determine where the page's data should be written.

The last of the operating systems we examined, HP-UX, takes a completely different stance on the issue of how to cache filesystem data. HP-UX continues to store file data in both the buffer cache and the page cache, though it does avoid the extra of copying of data that is present in pre-UBC NetBSD by reading data from disk directly into the page cache. The reasoning behind this is apparently that most files are only accessed by either `read()/write()` or `mmap()`, but not both, so as long as both mechanisms perform well individually, there's no need to redesign HP-UX just to fix the coherency issue. There is some attempt made to avoid incoherency between the two caches, but locking constraints prevent this from being completely effective.

There are other operating systems which have implemented a unified cache (eg. Compaq's Tru64 UNIX and IBM's AIX), but we were unable to find information on the design of these operating systems for comparison.

5 Performance

Since UBC is unfortunately not yet finished, a detailed performance analysis would be premature. However, we have made some simple comparisons just to see where we stand. The hardware used for this test was a 333MHz Pentium II with 64MB of RAM and a 12GB IDE disk. The operations performed were a series of "dd" commands designed to expose the behaviour of sequential reads and writes. We create a 1GB file (which is much larger than the physical memory available for caching), then overwrite this file to see the speed at which the data modifications caused by the `write()` are flushed to disk without the overhead of allocating blocks to the file. Then we read back the entire file to get an idea of how fast the filesystem can get data from the disk. Finally, we read the first 50MB of the file (which should fit entirely in physical memory) several times to determine the speed of access to cached data. See Table 1 for the results of these tests.

The great disparity in the results of the first four tests on the three non-UBC operating systems is due to differences in performance of their IDE disk drivers. All of the operating systems tested except NetBSD with UBC do sequential buffered reads from a large file at the same speed as reads from the raw device, so all we can really say from this is that the other caching designs don't add any noticeable overhead. For reads, the UBC system is not yet running at device speed, so there's still room for improvement. Further analysis is required to determine the cause of the slowdown.

UBC obviously needs much improvement in the area of write performance. This is partly due to UVM not being very efficient about flushing modified pages when memory is low and partly because the filesystem code currently doesn't trigger any asynchronous writes to disk during a big sequence of writes, so the writes to disk are all started by the inefficient UVM code. We've been concentrating on read performance so far, so this poor write performance is not surprising.

The interesting part of this test series is the set of tests where we read the same 50MB file five times. This clearly shows the benefit of the increased memory available for caching in the UBC system over NetBSD without UBC. In NetBSD 1.4.2, all five reads occurred at the speed of the device, whereas in all the other systems the reads were completed at memory speed after several runs. We have no explanation for why FreeBSD and Linux didn't complete the second 50MB read at memory speed, or why Linux didn't complete even the third read

Experiment			Run Time (seconds)			
Input	Output	Size	NetBSD 1.4.2	NetBSD with UBC	FreeBSD 3.4	Linux 2.2.12-20smp
raw device	/dev/null	1GB	72.8	72.7	279.3	254.6
/dev/zero	new file	1GB	83.8	193.0	194.3	163.9
/dev/zero	overwrite file	1GB	79.4	186.6	192.2	167.3
non-resident file	/dev/null	1GB	72.7	86.7	279.3	254.5
non-resident file	/dev/null	50MB	3.6	4.3	13.7	12.8
resident file	/dev/null	50MB	3.6	0.8	4.1	11.5
repeat above	/dev/null	50MB	3.6	0.8	0.7	4.5
repeat above	/dev/null	50MB	3.6	0.8	0.7	0.8
repeat above	/dev/null	50MB	3.6	0.8	0.7	0.8

Table 1: UBC performance comparison.

at memory speed.

6 Conclusion

In this paper we introduced UBC, a improved design for filesystem and virtual memory caching in NetBSD. This design includes many improvements over the previous design used in NetBSD by:

- Eliminating double caching of file data in the kernel (and the possibility of cache incoherency that goes with it) when the same file is accessed via different interfaces.
- Allowing more flexibility in how physical memory is used, which can greatly improve performance for applications whose data fits in physical memory.

7 Availability

This work will be part of a future release of NetBSD once it is completed. Until then, the source code is available in the “chs-ubc2” branch in the NetBSD CVS tree, accessible via anonymous CVS. See <http://www.netbsd.org/Sites/net.html> for details.

This being a work-in-progress, there is naturally much more work to do! Planned work includes:

- Integration of UBC into the NetBSD development source tree and performance improvement. The pagedaemon needs to be enhanced to deal with the

much larger amount of page-cache data which will be dirty.

- Elimination of the data copying in `read()` and `write()` via UVM page loanout when possible. This could be done without UBC too, but with UBC it will be zero-copy instead of one-copy (from buffer cache to page cache).
- Elimination of the need to map pages to do I/O to them by adding a page list to `struct buf` and adding glue in `bus_dma` to map pages temporarily for hardware that actually needs that.
- Adding support for “XIP” (eXecute In Place). This will allow zero-copy access to filesystem images stored in flash roms or other memory-mapped storage devices.
- Adding support for cache coherency in layered filesystems. (The current UBC design does not address caching in layered filesystems.)

Acknowledgments

We would like to thank everyone who helped review drafts of this paper. Special thanks to Chuck Cranor!

References

- [1] The NetBSD Project. The NetBSD Operating System. See <http://www.netbsd.org/> for more information.
- [2] C. Cranor and G. Parulkar. The UVM Virtual Memory System. In *Proceedings of the 1999 USENIX Technical Conference*, June 1999.

- [3] Marice J. Bach. The Design of the UNIX Operating System. Prentice Hall, February 1987.
- [4] J. Moran, R. Gingell and W. Shannon. Virtual Memory Architecture in SunOS. In *Proceedings of USENIX Summer Conference*, pages 81-94. USENIX, June 1987.
- [5] J. Moran. SunOS Virtual Memory Implementation. In *Proceedings of the Spring 1988 European UNIX Users Group Conference*, April 1988.
- [6] The FreeBSD Project. The FreeBSD Operating System. See <http://www.freebsd.org/> for more information.
- [7] L. Torvalds, et al. The Linux Operating System. See <http://www.linux.org/> for more information.

Mbuf issues in 4.4BSD IPv6/IPsec support — experiences from KAME IPv6/IPsec implementation —

Jun-ichiro itojun Hagino

KAME Project
Research Laboratory, Internet Initiative Japan Inc.
<http://www.kame.net/>
itojun@iijlab.net

ABSTRACT

The 4.4BSD network stack has made certain assumptions regarding the packets it will handle. In particular, 4.4BSD assumes that (1) the total protocol header length is shorter than or equal to MHLEN, usually 100 bytes, and (2) there are a limited number of protocol headers on a packet. Neither of these assumptions hold any longer, due to the way IPv6/IPsec specifications are written.

We at the KAME project are implementing IPv6 and IPsec support code on top of 4.4BSD. To cope with the problems, we have introduced the following changes: (1) a new function called *m_pulldown*, which adjusts the mbuf chain with a minimal number of copies/allocations, and (2) a new calling sequence for parsing inbound packet headers. These changes allow us to manipulate incoming packets in a safer, more efficient, and more spec-conformant way. The technique described in this paper is integrated into the KAME IPv6/IPsec stack kit, and is freely available under BSD copyright. The KAME codebase is being merged into NetBSD, OpenBSD and FreeBSD. An integration into BSD/OS is planned.

1. 4.4BSD incompatibility with IPv6/IPsec packet processing

The 4.4BSD network code holds a packet in a chain of “mbuf” structures. Each mbuf structure has three flavors:

- non-cluster header mbuf, which holds MHLEN (100 bytes in a 32bit architecture installation of 4.4BSD),
- non-cluster data mbuf, which holds MLEN (104 bytes), and
- cluster mbuf which holds MCLBYTES (2048 bytes).

We can make a chain of mbuf structures as a linked list. Mbuf chains will efficiently hold variable-length packet data. Such chains also enable us to insert or remove some of the packet data from the chain without data copies.

When processing inbound packets, 4.4BSD uses a function called *m_pullup* to ease the manipulation of data content in the mbufs. It also uses a deep function call tree for inbound packet

processing. While these two items work just fine for traditional IPv4 processing, they do not work as well with IPv6 and IPsec processing.

1.1. Restrictions in 4.4BSD *m_pullup*

For input packet processing, the 4.4BSD network stack uses the *m_pullup* function to ease parsing efforts by adjusting the data content in mbufs for placement onto the continuous memory region. *m_pullup* is defined as follows:

```
struct mbuf *  
m_pullup(m, len)  
{  
    struct mbuf *m;  
    int len;  
    ...  
}
```

m_pullup will ensure that the first *len* bytes in the packet are placed in the continuous memory region. After a call to *m_pullup*, the caller can safely access the the first *len* bytes of the packet, assuming that they are continuous. The caller can, for example, safely use pointer variables into the continuous region, as long as they point inside the *len* boundary.

IPv6 header next = routing	routing header next = auth	auth header next = TCP	TCP header	TCP payload
-------------------------------	-------------------------------	---------------------------	------------	-------------

Figure 1: IPv6 extension header chain

m_pullup makes certain assumptions regarding protocol headers. *m_pullup* can only take *len* upto MHLEN. If the total packet header length is longer than MHLEN, *m_pullup* will fail, and the result will be a loss of the packet. Under IPv4 (Postel, 1981), the length assumption worked fine in most cases, since for almost every protocol, the total length of the protocol header part was less than MHLEN. Each packet has only two protocol headers, including the IPv4 header. For example, the total length of the protocol header part of a TCP packet (up to TCP data payload) is a maximum of 120 bytes. Typically, this length is 40 to 48 bytes. When an IPv4 option is present, it is stripped off before TCP header processing, and the maximum length passed to *m_pullup* will be 100.

- 1 The IPv4 header occupies 20 bytes.
- 2 The IPv4 option occupies 40 bytes maximum. It will be stripped off before we parse the TCP header. Also note that the use of IPv4 options is very rare.
- 3 The TCP header length is 20 bytes.
- 4 The TCP option is 40 bytes maximum. In most cases it is 0 to 8 bytes.

IPv6 specification (Deering, 1998) and IPsec specification (Kent, 1998) allow more flexible use of protocol headers by introducing chained extension headers. With chained extension headers, each header has a "next header field" in it. A chain of headers can be made as shown in Figure 2. The type of protocol header is determined by inspecting the previous protocol header. There is no restriction in the number of extension headers in the spec.

Because of extension header chains, there is now no upper limit in protocol packet header length. The *m_pullup* function would impose unnecessary restriction to the extension header processing. In addition, with the introduction of IPsec, it is now impossible to strip off extension headers during inbound packet processing. All of the data on the packet must be retained if it is to be authenticated using Authentication Header (Kent, 1998). Continuing the use of *m_pullup* will limit the number of extension headers

allowed on the packet, and could jeopardize the possible usefulness of IPv6 extension headers.¹

Another problem related to *m_pullup* is that it tends to copy the protocol header even when it is unnecessary to do so. For example, consider the mbuf chain shown in Figure 2:

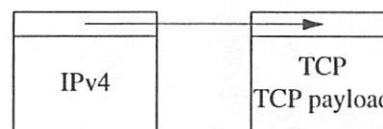


Figure 2: mbuf chain before *m_pullup*

Here, the first mbuf contains an IPv4 header in the continuous region, and the second mbuf contains a TCP header in the continuous region. When we look at the content of the TCP header, under 4.4BSD the code will look like the following:

```
struct ip *ip;
struct tcphdr *th;
ip = mtod(m, struct ip *);
/* extra copy with m_pullup */
m = m_pullup(m, iphdrln + tcphdrln);
/* MUST reinit ip */
ip = mtod(m, struct ip *);
th = mtod(m, caddr_t) + iphdrln;
```

As a result, we will get a mbuf chain shown in Figure 3.

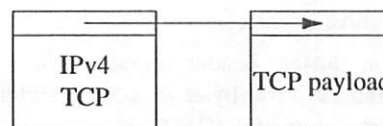


Figure 3: mbuf chain in figure 2 after *m_pullup*

Because *m_pullup* is only able to make a continuous region starting from the top of the mbuf chain, it copies the TCP portion in second mbuf into the first mbuf. The copy could be avoided if *m_pullup* were clever enough to handle this case. Also, the caller side is required to reinitialize all of the pointers that point to the content of mbuf, since after *m_pullup*, the first mbuf on the chain

¹ In IPv4 days, the IPv4 options turned out to be unusable due to a lack of implementation. This was because most commercial products simply did not support IPv4 options.

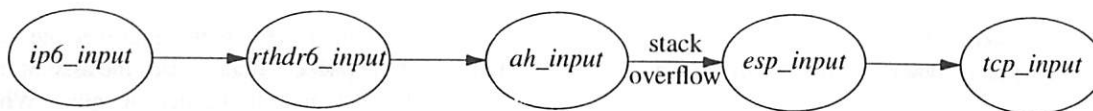


Figure 5: an excessively deep call chain can cause kernel stack overflow

can be reallocated and lives at a different address than before. While *m_pullup* design has provided simplicity in packet parsing, it is disadvantageous for protocols like IPv6.

The problems can be summarized as follows: (1) *m_pullup* imposes too strong restriction on the total length of the packet header (MHLEN); (2) *m_pullup* makes an extra copy even when this can be avoided; and (3) *m_pullup* requires the caller to reinitialize all of the pointers into the mbuf chain.

1.2. Protocol header processing with a deep function call chain

Under 4.4BSD, protocol header processing will make a chain of function calls. For example, if we have an IPv4 TCP packet, the following function call chain will be made (see Figure 4):

- (1) *ipintr* will be called from the network software interrupt logic,
- (2) *ipintr* processes the IPv4 header, then calls *tcp_input*.
- (3) *tcp_input* will process the TCP header and pass the data payload to the socket queues.

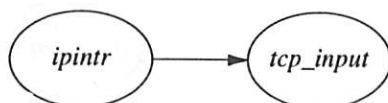


Figure 4: function call chain in IPv4 inbound packet processing

If chained extension headers are handled as described above, the kernel stack can overflow by a deep function call chain, as shown in Figure 5. IPv6/IPsec specifications do not define any upper limit to the number of extension headers on a packet, so a malicious party can transmit a “legal” packet with a large number of chained headers in order to attack IPv6/IPsec implementations. We have experienced kernel stack overflow in IPsec code, tunnelled packet processing code, and in several other cases. The IPsec processing routines tend to use a large chunk of memory on the kernel stack, in order to hold intermediate data and the secret keys used for encryption.² We

cannot put the intermediate data region into a static data region outside of the kernel stack, because it would become a source of performance drawback on multiprocessors due to data locking.

Even though the IPv6 specifications do not define any restrictions on the number of extension headers, it may be possible to impose additional restriction in an IPv6 implementation for safety. In any case, it is not possible to estimate the amount of the kernel stack, which will be used by protocol handlers. We need a better calling convention for IPv6/IPsec header processing, regardless of the limits in the number of extension headers we may impose.

2. KAME approach

This section describes the approaches we at the KAME project took against the problems mentioned in the previous section. We introduce a new function called *m_pulldown*, in place of *m_pullup*, for adjusting payload data in the mbuf. We also change the calling sequence for the protocol input function.

2.1. What is the KAME project?

In the early days of IPv6/IPsec development, the Japanese research community felt it very important to make a reference code available in a freely-redistributable form for educational, research and deployment purposes. The KAME project is a consortium of 7 Japanese companies and an academic research group. The project aims to deliver IPv6/IPsec reference implementation for 4.4BSD, under BSD license. The KAME project intends to deliver the most spec-conformant IPv6/IPsec implementation possible.

2.2. *m_pulldown* function

Here we introduce a new function, *m_pulldown*, to address the 3 problems with *m_pullup* that we have described above. The actual source code is included at the end of this paper. The function prototype is as follows:

² For example, blowfish encryption processing code typically uses an intermediate data region of 4K or more. With typical 4.4BSD installation on i386 architecture, the kernel stack region occupies less than 8K bytes and does not grow on demand.

```

struct mbuf *
m_pullup(m, off, len, offp)
    struct mbuf *m;
    int off, len;
    int *offp;

```

m_pullup will ensure that the data region in the mbuf chain, starting at *off* and ending at *off + len*, is put into a continuous memory region. *len* must be smaller than, or equal to, MCLBYTES (2048 bytes). The function returns a pointer to an intermediate mbuf in the chain (we refer to the pointer as *n*), and puts the new offset in *n* to **offp*. If *offp* is NULL, the resulting region can be located by *mtod(n, caddr_t)*; if *offp* is non-null, it will be located at *mtod(n, caddr_t) + *offp*. The mbuf prior to *off* will remain untouched, so it is safe to keep the pointers to the mbuf chain. For example, consider the mbuf chain on Figure 6 as the input.

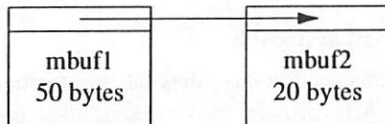


Figure 6: mbuf chain before the call to *m_pullup*. If we call *m_pullup* with *off* = 40, *len* = 10, and a non-null *offp*, the mbuf chain will remain unchanged. The return value will be a pointer to mbuf1, and **offp* will be filled with 40. If we call *m_pullup* with *off* = 40, *len* = 20, and null *offp*, then the mbuf chain will be modified as shown in Figure 7, by allocating a new mbuf, mbuf3, into the middle and moving data from both mbuf1 and mbuf2. The function returns a pointer to mbuf3.

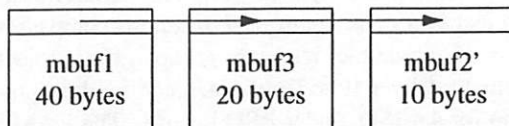


Figure 7: mbuf chain after call to *m_pullup*, with *off* = 40 and *len* = 20

The *m_pullup* function solves all 3 problems in *m_pullup* that were described in the previous section. *m_pullup* does not copy mbufs when copying is not necessary. Since it does not modify the mbuf chain prior to the specified offset *off*, it is not necessary for the caller to re-initialize the pointers into the mbuf data region. With *m_pullup*, we always needed to specify the data payload length, starting from the very first byte in the packet. With *m_pullup*, we pass *off* as the offset to the data payload we are interested in. This change avoids extra data manipulation when

we are only interested in the intermediate data portion of the packet. It also eases the assumption regarding total packet header length. While *m_pullup* assumes that the total packet header length is smaller than or equal to MHLEN (100 bytes), *m_pullup* assumes that single packet header length is smaller than or equal to MCLBYTES (2048 bytes). With mbuf framework this is the best we can do, since there is no way to hold continuous region longer than MCLBYTES in a standard mbuf chain.

2.3. New function prototype for inbound packet processing

For IPv6 processing, our code does not make a deep function call chain. Rather, we make a loop in the very last part of *ip6_input*, as shown in Figure 8. IPPROTO_DONE is a pseudo-protocol type value that identifies the end of the extension header chain. If more protocol headers exist, each header processing code will update the pointer variables and return the next extension header type. If the final header in the chain has been reached, IPPROTO_DONE is returned. With this code, we no longer have a deep call chain for IPv6/IPsec processing. Rather, *ip6_input* will make calls to each extension header processor directly. This avoids the possibility of overflowing the kernel stack due to multiple extension header processing.

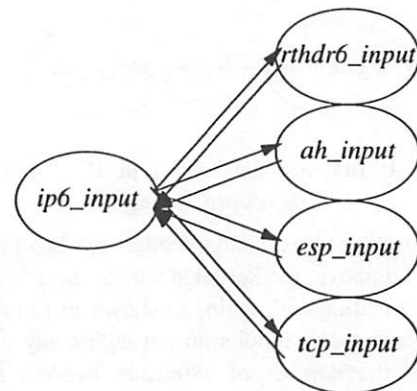


Figure 9: KAME avoids function call chain by making a loop in *ip6_input*

Regardless of the calling sequence imposed by the *pr_input* function prototype, it is important not to use up the kernel stack region in protocol handlers. Sometimes it is necessary to decrease the size of kernel stack usage by using pointer variables and dynamically allocated regions.

```

struct ip6protosw {
    int (*pr_input) __P((struct mbuf **, int *, int));
    /* and other members */
};

ip6_input(m)
    struct mbuf *m;
{
    /* in the very last part */
    extern struct ip6protosw inet6sw[];
    /* the first one in extension header chain */
    nxt = ip6.ip6_nxt;
    while (nxt != IPPROTO_DONE)
        nxt = (*inet6sw[ip6_protox[nxt]].pr_input)(&m, &off, nxt);
}

/* in each header processing code */
int
foohdr_input(mp, offp, proto)
    struct mbuf **mp;
    int *offp;
    int proto;
{
    /* some processing, may modify mbuf chain */

    if (we have more header to go) {
        *mp = newm;
        *offp = nxtoff;
        return nxt;
    } else {
        m_freem(newm);
        return IPPROTO_DONE;
    }
}

```

Figure 8: KAME IPv6 header chain processing code.

3. Alternative approaches

Many BSD-based IPv6 stacks have been implemented. While the most popular stacks include NRL, INRIA and KAME, dozens of other BSD-based IPv6 implementations have been made. This section presents alternative approaches for purposes of comparison.

3.1. NRL *m_pullup2*

The latest NRL IPv6 release copes with the *m_pullup* limitation by introducing a new function, *m_pullup2*. *m_pullup2* works similarly to *m_pullup*, but it allows *len* to extend up to MCLBYTES, which corresponds to 2048 bytes in a typical installation. When the *len* parameter is smaller than or equal to MHLEN, *m_pullup2* simply calls *m_pullup* from the inside.

While *m_pullup2* works well for packet headers up to MCLBYTES with very little change in code, it does not avoid making unnecessary copies. It also imposes restrictions on the total length of packet headers. The assumption here is that the total length of packet headers is less than MCLBYTES.

3.2. Hydrangea changes to *m_devget*

The Hydrangea IPv6 stack was implemented by a group of Japanese researchers, and is one of the ancestors of the KAME IPv6 stack. The Hydrangea IPv6 stack avoids the need for *m_pullup* by modifying the mbuf allocation policy in drivers. For inbound packets, the drivers allocate mbufs by using the *m_devget* function, or by re-implementing the behavior of *m_devget*. *m_devget* allocates mbuf as follows:

- 1 If the packet fits in MHLEN (100 bytes), allocate a single non-cluster mbuf.
- 2 If the packet is larger than MHLEN but fits in MHLEN + MLEN (204 bytes), allocate two non-cluster mbufs.
- 3 Otherwise, allocate multiple cluster mbufs, MCLBYTES (2048 bytes) in size.

For typical packets, the second case is where *m_pullup* is used. The Hydrangea stack avoids the use of *m_pullup* by eliminating the second case.

This approach worked well in most cases, but failed for (1) loopback interface, (2) tunnelled packets, and (3) non-conforming drivers. With the Hydrangea approach, every device driver had to be examined to ensure the new mbuf allocation

policy. We could not be sure if the constraint was guaranteed until we checked the driver code, and the Hydrangea approach raised many support issues. This was one of our motivations for introducing *m_pulldown*.

4. Comparisons

This section compares the following three approaches in terms of their characteristics and actual behavior: (1) 4.4BSD *m_pullup*, (2) NRL *m_pullup2*, and (3) KAME *m_pulldown*.

4.1. Comparison of assumption

Table 1 shows the assumptions made by each of the three approaches. As mentioned earlier, *m_pullup* imposes too stringent requirement for the total length of packet headers. *m_pullup2* is workable in most cases, although this approach adds more restrictions than the specification claims. *m_pulldown* assumes that the single packet header is smaller than MCLBYTES, but makes no restriction regarding the total length of packet headers. With a standard mbuf chain, this is the best *m_pulldown* can do, since there is no way to hold continuous region longer than MCLBYTES. This characteristic can contribute to better specification conformance, since *m_pull-down* will impose fewer additional restrictions due to the requirements of implementation.

Among the three approaches, only *m_pull-down* avoids making unnecessary copies of intermediate header data and avoids pointer reinitialization after calls to these functions. These attributes result in smaller overhead during input packet processing.

At present, we know of no other 4.4BSD-based IPv6/IPsec stack that addresses kernel stack overflow issues, although we are open to new perspectives and new information.

4.2. Performance comparison based on simulated statistics

To compare the behavior and performance of *m_pulldown* against *m_pullup* and *m_pullup2* using the same set of traffic and mbuf chains, we have gathered simulated statistics for *m_pullup* and *m_pullup2*, in *m_pulldown* function. By running a kernel using the modified *m_pulldown* function, we can easily gather statistics for these three functions against exactly the same traffic.

The comparison was made on a computer (with Celeron 366MHz CPU, 192M bytes of

memory) running NetBSD 1.4.1 with the KAME IPv6/IPsec stack. Network drivers allocate mbufs just as normal 4.4BSD does. *m_pulldown* is called whenever it is needed to ensure continuity in packet data during inbound packet processing. The role of the computer is as an end node, not a router.

To describe the content of the following table, we must look at the source code fragment. Figure 10 shows the code fragment from our source code. The code fragment will (1) make the TCP header on the mbuf chain *m* at offset *hdrlen* continuous, and (2) point the region with pointer *th*. We use a macro named *IP6_EXTHDR_CHECK*, and the code before and after the macro expansion is shown in the figure.

```
/* ensure that *th from hdrlen is continuous */
/* before macro expansion... */
struct tcphdr *th;
IP6_EXTHDR_CHECK(th, struct tcphdr *, m,
                 hdrlen, sizeof(*th));
if (th == NULL)
    return; /*m is already freed*/

/* after macro expansion... */
struct tcphdr *th;
int off;
struct mbuf *n;
if (m->m_len < hdrlen + sizeof(*th)) {
    n = m_pulldown(m, hdrlen, sizeof(*th), &off);
    if (n)
        th = (struct tcphdr *) (mtod(n, caddr_t) + off);
    else
        th = NULL;
} else
    th = (struct tcphdr *) (mtod(m, caddr_t) + hdrlen);
if (th == NULL)
    return;
```

Figure 10: code fragment for trimming mbuf chain.

In Table 2, the first column identifies the test case. The second column shows the number of times the *IP6_EXTHDR_CHECK* macro was used. In other words, it shows the number of times we have made checks against mbuf length. The remaining columns show, from left to right, the number of times memory allocation/copy was performed in each of the variants. In the case of *m_pullup*, we counted the number of cases we passed *len* in excess of MHLEN (96 bytes in this installation). This result suggests that there was no packet with a packet header portion larger than MCLBYTES (2048 bytes). In the evaluation we have used *m_pulldown* against IPv6 traffic only.

	<i>m_pullup</i>	<i>m_pullup2</i>	<i>m_pulldown</i>
total header length	MHLEN(100)	MCLBYTES(2048)	—
single header length	—	—	MCLBYTES(2048)
avoids copy on intermediate headers	no	no	yes
avoids pointer reinitialization	no	no	yes

Table 1: assumptions in mbuf manipulation approaches.

test	len checks	<i>m_pulldown</i>			<i>m_pullup</i>			<i>m_pullup2</i>	
		call	alloc	copy	alloc	copy	fail	alloc	copy
(1)	204923	1706	1595	1596	165	165	1541	1596	1596
(2)	1063995	23786	22931	23008	1171	1229	22557	22895	22953
(3)	520028	1245	948	957	432	432	813	945	945
(4)	438602	180	6	6	178	178	2	24	24
(5)	5570	2236	206	206	812	812	1424	1424	1424

Table 2: number of mbuf allocation/copy against traffic

test	IPv6 input	TCP	UDP	ICMPv6	1 mbuf	2 mbufs	ext mbuf(s)
(1)	29334	20892	2699	5739	3624	15632	10078
(2)	313218	215919	15930	80263	38751	172976	101491
(3)	132267	117822	8561	5882	12782	59799	59686
(4)	73160	66512	5249	1343	7475	42053	23632
(5)	1433	148	53	52	103	1203	127

Table 3: Traffic characteristics for tests in Table 2

From these measured results, we obtain several interesting observations. *m_pullup* actually failed on IPv6 traffic. If an IPv6 implementation uses *m_pullup* for IPv6 input processing, it must be coded carefully so as to avoid trying *m_pullup* against any length longer than MHLEN. To achieve this end, the code copies the data portion from the mbuf chain to a separate buffer, and the cost of memory copies becomes a penalty.

Due to the nature of this simulation, the comparison described above may contain an implicit bias. Since the IPv6 protocol processing code is written by using *m_pulldown*, the code is somewhat biased toward *m_pulldown*. If a programmer had to write the entire IPv6 protocol processing with *m_pullup* only, he or she would use *m_copydata* to copy intermediate extension headers buried deep inside the header chains, thus making it unnecessary to call *m_pullup*. In any case, a call to *m_copydata* will result in a data copy, which causes extra overhead.

In all cases, the number of length checks (second column) exceeds the number of inbound packets. This behavior is the same as in the original 4.4BSD stack; we did not add a significant number of length checks to the code. This is because *m_pulldown* (or *m_pullup* in the 4.4BSD case) is called as necessary during the parsing of

the headers. For example, to process a TCP-over-IPv6 packet, at least 3 checks would be made against *m->m_len*; these checks would be made to grab the IPv6 header (40 bytes), to grab the TCP header (20 bytes), and to grab the TCP header and options (20 to 60 bytes). The length of the TCP option part is kept inside the TCP header, so the length needs to be checked twice for the TCP part.

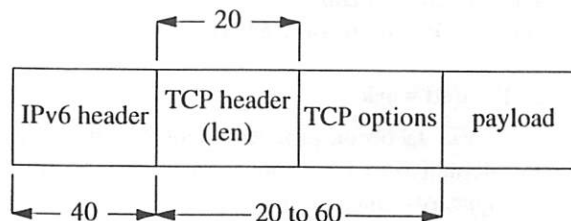


Figure 11: processing a TCP-over-IPv6 packet requires 3 length checks.

The results suggest that we call *m_pulldown* more frequently in ICMPv6 processing than in the processing of other protocols. These additional calls are made for parsing of ICMPv6 and for neighbor discovery options. The use of loopback interface also contributes to the use of *m_pulldown*.

In the tests, the number of copies made in the *m_pullup2* case is similar to the number made

in the *m_pulldown* case. *m_pulldown* makes less copies than *m_pullup2* against packets like below:

- A packet is kept in multiple mbuf. With mbuf allocation policy in *m_devget*, we will see two mbufs to hold single packet if the packet is larger than MHLEN and smaller than MHLEN + MLEN.
- We have extension headers in both mbufs. Header portion in the packet needs to occupy first mbuf and part of subsequent mbufs.

To present the difference, we have generated an IPv6 packet with a routing header, with 4 IPv6 addresses. The test result is presented as the 5th test in the table. Packet will look like Figure 12. First 112 bytes are occupied by an IPv6 header and a routing header, and the remaining 16 bytes are used for an ICMPv6 header and payload. The packet met the above condition, and *m_pulldown* made less copies than *m_pullup2*. To process single incoming ICMPv6 packet shown in the figure, *m_pullup2* made 7 copies while *m_pulldown* made only 1 copy.

```
node A (source) = 2001:240:0:200:260:97ff:fe07:69ea
node B (destination) = 2001:240:0:200:a00:5aff:fe38:6f86
17:39:43.346078 A > B:
  srct (type=0, segleft=4, [0]B, [1]B, [2]B, [3]B):
    icmp6: echo request (len 88, hlim 64)
      6000 0000 0058 2b40 2001 0240 0000 0200
      0260 97ff fe07 69ea 2001 0240 0000 0200
      0a00 5aff fe38 6f86 3a08 0004 0000 0000
      2001 0240 0000 0200 0a00 5aff fe38 6f86
      2001 0240 0000 0200 0a00 5aff fe38 6f86
      2001 0240 0000 0200 0a00 5aff fe38 6f86
      2001 0240 0000 0200 0a00 5aff fe38 6f86
      8000 b650 030e 00c8 ce6e fd38 d553 0700
```

Figure 12: Packets with IPv6 routing header.

During the test, we experienced no kernel stack overflow, thanks to a new calling sequence between IPv6 protocol handlers.

5. Related work

Van Jacobson proposed pbuf structure³ as an alternative to BSD mbuf structure. The proposal has two main arguments. First is the use of continuous data buffer, instead of chained fragments like mbufs. Another is the improvement to TCP performance by restructuring TCP input/output handling. While the latter point still holds for IPv6, we believe that the former point must be reviewed carefully before being used with IPv6. Our experience suggests that we need to insert many intermediate extension headers into the packet data during IPv6 outbound packet processing. We believe that mbuf is more suitable than

³ A reference should be here, but I'm having hard time finding published literature for it.

the proposed pbuf structure for handling the packet data efficiently. Using pbuf may result in the making of more copies than in the mbuf case.

In a cross-BSD portability paper (Metz, 1999), Craig Metz described *nbuf* structure in NRL IPv6/IPsec stack. *nbuf* is a wrapper structure used to unify linux linear-buffer packet management and BSD mbuf structure, and is not closely related to the topic of this paper. The *m_pullup2* example discussed in this paper is drawn from the NRL implementation.

6. Conclusions

This paper discussed mbuf manipulation in a 4.4BSD-based IPv6/IPsec stack, namely KAME IPv6/IPsec implementation. 4.4BSD makes certain assumptions regarding packet header length and its format. For IPv6/IPsec support, we removed those assumptions from the 4.4BSD code. We introduced the *m_pulldown* function and a new function call sequence for inbound packet processing. These innovations helped us to implement IPv6/IPsec in a very spec-conformant manner, with fewer implementation restrictions added against specifications.

The described code is publically available, under a BSD-like license, at <ftp://ftp.kame.net/>. KAME IPv6/IPsec stack is being merged into 4.4BSD variants like FreeBSD, NetBSD and OpenBSD. An integration into BSD/OS is planned. We will be able to see official releases of these OSes with KAME code soon.

7. Acknowledgements

The paper was made possible by the collective efforts of researchers at the KAME project and the WIDE project and of other IPv6 implementers at large. We would also like to acknowledge all four BSD groups who helped us improve the KAME IPv6 stack code by sending bug reports and improvement suggestions, and the Freenix reviewers helped polish the paper.

References

- Postel, 1981.
John Postel, "Internet Protocol" in *RFC791* (September 1981). <ftp://ftp.isi.edu/in-notes/rfc791.txt>.
- Deering, 1998.
S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification" in

RFC2460 (December 1998).
<ftp://ftp.isi.edu/in-notes/rfc2460.txt>.

Kent, 1998.

Stephen Kent and Randall Atkinson, "Security Architecture for the Internet Protocol" in *RFC2401* (November 1998).
<ftp://ftp.isi.edu/in-notes/rfc2401.txt>.

Kent, 1998.

Stephen Kent and Randall Atkinson, "IP Authentication Header" in *RFC2402* (November 1998). <ftp://ftp.isi.edu/in-notes/rfc2402.txt>.

Metz, 1999.

Craig Metz, "Porting Kernel Code to Four BSDs and Linux" in *1999 USENIX annual technical conference, Freenix track* (June 1999). <http://www.usenix.org/publications/library/proceedings/usenix99/metz.html>.

```

/*
 * ensure that [off, off + len) is contiguous on the mbuf chain "m".
 * packet chain before "off" is kept untouched.
 * if offp == NULL, the target will start at <retval, 0> on resulting chain.
 * if offp != NULL, the target will start at <retval, *offp> on resulting chain.
 *
 * on error return (NULL return value), original "m" will be freed.
 *
 * XXX M_TRAILINGSPACE/M_LEADINGSPACE on shared cluster (sharedcluster)
 */
struct mbuf *
m_pullupdown(m, off, len, offp)
    struct mbuf *m;
    int off, len;
    int *offp;

    struct mbuf *n, *o;
    int hlen, tlen, olen;
    int sharedcluster;

/* check invalid arguments. */
if (m == NULL)
    panic("m == NULL in m_pullupdown()");
if (len > MCLBYTES) {
    m_freem(m);
    return NULL; /* impossible */
}

n = m;
while (n != NULL && off > 0) {
    if (n->m_len > off)
        break;
    off -= n->m_len;
    n = n->m_next;
}
/* be sure to point non-empty mbuf */
while (n != NULL && n->m_len == 0)
    n = n->m_next;
if (!n) {
    m_freem(m);
    return NULL; /* mbuf chain too short */
}

/*
 * the target data is on <n, off>.
 * if we got enough data on the mbuf "n", we're done.
 */
if ((off == 0 || offp) && len <= n->m_len - off)
    goto ok;

/*
 * when len < n->m_len - off and off != 0, it is a special case.
 * len bytes from <n, off> sits in single mbuf, but the caller does
 * not like the starting position (off).
 * chop the current mbuf into two pieces, set off to 0.
 */
if (len < n->m_len - off) {
    o = m_copy(m, n, off, n->m_len - off, M_DONTWAIT);
    if (o == NULL) {
        m_freem(m);
        return NULL; /* ENOBUFFS */
    }
    n->m_len = off;
    o->m_next = n->m_next;
    n->m_next = o;
    n = n->m_next;
    off = 0;
    goto ok;
}

/*
 * we need to take hlen from <n, off> and tlen from <n->m_next, 0>,
 * and construct contiguous mbuf with m_len == len.
 * note that hlen + tlen == len, and tlen > 0.
 */
hlen = n->m_len - off;
tlen = len - hlen;

/*
 * ensure that we have enough trailing data on mbuf chain.
 * if not, we can do nothing about the chain.
 */
olen = 0;
for (o = n->m_next; o != NULL; o = o->m_next)
    olen += o->m_len;
if (hlen + olen < len) {
    m_freem(m);
    return NULL; /* mbuf chain too short */
}

/*
 * easy cases first.
 * we need to use m_copydata() to get data from <n->m_next, 0>.
 */
if ((n->m_flags & M_EXT) == 0)
    sharedcluster = 0;
else {
    if (n->m_ext.ext_free)
        sharedcluster = 1;
    else if (MCLISREFERENCED(n))
        sharedcluster = 1;
    else
        sharedcluster = 0;
}
if ((off == 0 || offp) && M_TRAILINGSPACE(n) >= tlen
    && !sharedcluster) {
    m_copydata(n->m_next, 0, tlen, mtd(n, caddr_t) + n->m_len);
    n->m_len += tlen;

```

```

    m_adj(n->m_next, tlen);
    goto ok;
}
if ((off == 0 || offp) && M_LEADINGSPACE(n->m_next) >= hlen
    && !sharedcluster) {
    n->m_next->m_data -= hlen;
    n->m_next->m_len += hlen;
    bcopy(mtd(n, caddr_t) + off, mtd(n->m_next, caddr_t), hlen);
    n->m_len -= hlen;
    n = n->m_next;
    off = 0;
    goto ok;
}

/*
 * now, we need to do the hard way. don't m_copy as there's no room
 * on both end.
 */
MGET(o, M_DONTWAIT, m->m_type);
if (o == NULL) {
    m_freem(m);
    return NULL; /* ENOBUFFS */
}
if (len > MHLEN) { /* use MHLEN just for safety */
    MCLGET(o, M_DONTWAIT);
    if ((o->m_flags & M_EXT) == 0) {
        m_freem(m);
        m_free(o);
        return NULL; /* ENOBUFFS */
    }
}
/* get hlen from <n, off> into <o, 0> */
o->m_len = hlen;
bcopy(mtd(n, caddr_t) + off, mtd(o, caddr_t), hlen);
n->m_len -= hlen;
/* get tlen from <n->m_next, 0> into <o, hlen> */
m_copydata(n->m_next, 0, tlen, mtd(o, caddr_t) + o->m_len);
o->m_len += tlen;
m_adj(n->m_next, tlen);
o->m_next = n->m_next;
n->m_next = o;
n = o;
off = 0;

ok:
if (offp)
    *offp = off;
return n;
}

```

malloc() Performance in a Multithreaded Linux Environment

Chuck Lever and David Boreham, *Sun-Netscape Alliance*

*Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor*

linux-scalability@citi.umich.edu
<http://www.citi.umich.edu/projects/linux-scalability>

Abstract

Network servers make special demands that other types of applications may not make on memory allocators. We describe a simple malloc() microbenchmark suite that tests the ability of malloc() to divide its work efficiently among multiple threads and processors. The purpose of this suite is to determine the suitability of an operating system's heap allocator for use with network servers running in an SMP environment.

1. Introduction

Modern network servers often employ multithreading to leverage multi-CPU hardware and to increase I/O concurrency. As network services scale to tens of thousands of clients per server, they depend on the ability of the underlying operating system and vendor-provided library routines to support multithreading efficiently.

The application-level memory allocator, or heap allocator, is a system API that must scale well with the number of application threads and the number of processors in the system. Known in UNIX as malloc(), the heap allocator makes use of several important system facilities, including mutex locking and virtual memory page allocation. Analyzing the performance of malloc() in a multithreaded and multi-CPU environment can provide important information about potential system inefficiency. Finding ways to improve the performance of malloc() can benefit the performance of any sophisticated multithreaded application, such as network servers.

Network servers make special demands that other types of applications may not make on memory allocators [5]. In this report we describe a simple malloc() microbenchmark suite that drives multithreaded loads to test the ability of malloc() to divide its work efficiently among multiple threads and processors. The purpose of this suite is to determine the suitability of an operating system's heap allocator for use with network servers running in an SMP environment. We discuss initial results of the benchmarks, and show that malloc() performance is important to overall network server scalability.

2. Motivation for studying malloc()

Larson and Krishnan give a good general description of network server applications and specifically, how they interact with a system's heap allocator [5]. Network servers are generally large long-running applications that employ multithreading and asynchronous I/O. They handle many small requests on behalf of client applications connected via a network such as TCP/IP, maintain some amount of state per connected client, and are often required to maintain low latency, high data throughput, and predictable response time. Unlike most test applications used in traditional memory allocator studies [8], network servers experience a potentially un-

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

This document is Copyright © 2000 AOL-Netscape, Inc. Trade-marked material referenced in this document is copyright by its respective owner.

bounded input set of unpredictable requests rather than a finite input set.

The iPlanet directory server product is typical of many network server applications. It is a single multithreaded process that handles concurrent requests from many clients connected via TCP/IP. This software is often deployed on SMP hardware to take advantage of the potential scalability of multiple processors.

Using non-intrusive program counter sampling tools, iPlanet developers profiled a directory server running under several standard operating systems on four-processor machines. A typical workload in this environment causes classic lock contention symptoms when more than one processor is enabled, such as:

- Decrease in application performance,
- Increase in kernel mode time, and
- Significant time spent in the thread scheduler and in the operating system's low level lock code

Further investigation indicated that the principle source of contention is the heap allocator. The directory server employs the operating system's per-process heap, which vendors often make thread-safe by adding a single lock to protect heap allocation logic.

Several alternative solutions were proposed.

1. Re-write the server to avoid making heavy use of the heap. After picking low hanging fruit, progress became very slow and many new bugs were introduced due to increased complexity.
2. Implement per-thread storage within the server. This technique had been used with success in other iPlanet server products. However those products had been designed from scratch with per-thread storage. It would be very costly to re-write a large existing application to use this technique.
3. Replace the operating system's heap allocator with an implementation that has more reasonable behavior in a multithreaded environment. This allows the server's code base to remain unchanged, offering considerable savings in time and stability compared with other options.

The directory server development team prototyped and tested option 3. The performance improvement exceeded a factor of six on four-processor hardware. Subsequently a commercially developed heap allocator with enhancements to eliminate lock contention was integrated into the final version of the product.

To further this effort we have created a set of simple, portable benchmark programs to assess the scalability of an operating system's heap allocator as it interacts with low-level operating system facilities. We focus on

simple benchmarks that treat each allocator as a black box, rather than use more complex trace-driven allocator simulations. Our tests are not meant to be scientific measurements of allocator performance, but rather to provide only an indication of relative acceptability. In our experience, simple benchmarks can uncover basic architectural limitations that make an allocator inappropriate for use with network server applications. Furthermore, simple benchmarks are generally more portable.

Traditional analysis of heap allocators has focused on efficient use of space and minimal CPU overhead [7]. Our study instead tests three areas of heap allocator behavior related to good performance and scalability of multithreaded network servers on *multiprocessor* hardware:

1. Multithread scalability

As we add physical processors and threads, heap contention has direct impact on network server scalability. Our first benchmark starts several threads that request and free memory from the heap allocator. On multiprocessor hardware, an ideal allocator would show linear speed-up for as many threads as there are processors in the system.

It is well known that synchronization primitives add significant overhead to lightweight algorithms such as heap allocators. Berger and Blumofe claim that a single lock added to the allocator can slow it down by as much as 50% on modern hardware [1].

2. Unbounded memory consumption

Allocating memory in one thread and freeing the same object in a different thread can cause some heap allocators to abandon areas of memory. We'd like to measure how much normal allocator operation fragments the heap over time. This test specifically targets memory fragmentation caused by the allocator running in a multithreaded environment, rather than by pathological application behavior.

Note that this is not a traditional way to analyze `malloc()`'s heap fragmentation. Many implementers have focused on space efficiency; *i.e.* the ability of an allocator to provide the greatest number of allocated objects for a given amount of virtual address space. In fact, for network servers, it is acceptable to allow *some* space or time inefficiency in trade for other benefits, such as reduced heap fragmentation over time. This means a multithreaded network server can run for longer periods without exhausting its memory space due to orphaned memory.

3. Cache-conscious data placement

Gunwald has shown the performance advantages of a cache-friendly allocator on modern SMP hardware [3]. In other words, the heap allocator can help reduce the effects of false cache line sharing and improve effective memory bandwidth by assigning object addresses with the specific characteristics of the CPU caches in mind. This is relevant to single processor hardware as well as SMP servers because modern CPUs rely more than ever on a memory hierarchy to bridge the gap between processor and memory speeds.

Careful placement of heap-allocated objects can also result in a lower application page fault rate. While the cost of cache misses has increased significantly as processor speeds outpace memory speeds, the impact of page faults on application performance has become even worse for similar reasons.

One of the most effective ways to reduce an application's page fault rate is to use a space-efficient heap allocator. However space-efficiency may not provide the highest cache-friendliness [4]. Another effective way to reduce page faults is to add more memory to the system. It is often more difficult to expand the size of CPU caches. Also, microprocessor cache designs are generally made less efficient (worse cache hit rate) by size and cost requirements. Cache-conscious libraries and application code offer one way to maximize the benefits of the CPU caches.

A heap allocator can employ two mechanisms to increase the effectiveness of the CPU caches and reduce false sharing that can cause wasted memory bandwidth on SMP hardware. First, a heap object shared among threads should never share a cache line with another heap object. Even small objects should reside in their own cache line, if practical. Wilson, and Johnstone *et al.*, show that most modern allocators cause little real fragmentation beyond that caused by aligning objects to large address boundaries, so larger alignments may be a practical approach to reducing false sharing among CPU caches [7, 4].

Second, the allocator should take as much advantage of temporal locality as possible. Gunwald postulates that objects allocated at the same time tend to be used and then freed together [3]. We expect this behavior to be especially relevant for thread memory allocation.

3. A look at glibc's `malloc()`

The study described in this paper was initiated as iPlanet developers were in the process of porting the iPlanet directory and messaging servers to Linux. Given their experience with heap allocators in other operating systems, they wanted to know how Linux's allocator

compared to others. In this section we examine the Linux application-level heap allocator in detail.

Modern distributions of Linux use glibc version 2.0 and 2.1 as their C library. Glibc's implementors have adopted Gloger's ptmalloc as the glibc implementation of `malloc()` [2]. Ptmalloc has many desirable properties, including multiple heaps to reduce contention among threads sharing a single C library invocation.

Ptmalloc, based on Doug Lea's original implementation of `malloc()` [6], had several goals, including improved portability, space and time utilization, and added tunable parameters to control allocation behavior. Gloger's update to Lea's original retains these desirable behaviors, adds good multithreading behavior, and features several nice debugging extensions. The C library is built on most Linux distributions with debugging extensions and tunability disabled, so it is necessary to rebuild the C library or pre-load a separate version of `malloc()` in order to take advantage of these features. Alternatively, an application can invoke `mallopt(3)` to enable some of these features.

Ptmalloc maintains a linked list of subheaps. To reduce lock contention, ptmalloc searches for the first unlocked subheap and grabs memory from it to fulfill a `malloc()` request. If ptmalloc doesn't find an unlocked heap, it creates a new one. This is a simple way to grow the number of subheaps as appropriate without adding complicated schemes for hashing on thread or processor ID, or maintaining workload statistics. However, there is no facility to shrink the subheap list and nothing stops the heap list from growing without bound. There are some (not infrequent) pathological cases where a producer thread allocates objects so often that it causes freeing threads to release objects into other subheaps, resulting in unbounded heap growth.

Ptmalloc makes use of both `mmap()` and `sbrk()` when allocating heap arenas. `Malloc()` uses `sbrk()` for allocation requests smaller than 32 pages, and `mmap()` for allocation requests larger than 32 pages. In general these system calls are essentially the same under the covers. Both use anonymous maps to provide large pageable areas of virtual memory to processes. `Sbrk()` can allocate only a fraction of the full virtual address space, however: `sbrk()` is not smart enough to allocate around pre-existing mappings, such as system libraries, that may appear in the middle of the address space. Later versions (post 2.1.3) of glibc have special logic to retry an arena allocation with `mmap()` if `sbrk()` fails. Kernel functions that use `sbrk()`, such as dynamic library loading, can also stop working if the application fills up its virtual address space.

Possible ways to help performance in this area include optimizing the allocation of anonymous maps and re-

ducing and amortizing the overhead of these system calls by having `malloc()` allocate subheaps in larger chunks. We have already provided a version of `sbrk()` for the Linux kernel that removes acquisition of the global kernel lock in most paths (see `mm/mmap.c` in Linux kernel versions 2.3.5 through 2.3.7). This allows `sbrk()` to outperform `mmap()` of anonymous pages in the general case. In addition, making `sbrk()` work more flexibly when a process's virtual address space becomes fragmented improves `malloc()` performance for applications like network servers and large databases that allocate large quantities of small objects. Finally, improving the mechanism by which the kernel memory manager locates free areas in a process's virtual address space would provide significant benefits as address spaces become crowded with heap and text areas, and maps.

4. Benchmark description

There are three microbenchmarks in this suite, each exploring a different set of heap allocator characteristics.

- Benchmark 1 examines the heap allocator's ability to use multiple threads and processors efficiently.
- Benchmark 2 focuses on the heap allocator's ability to prevent orphaned objects and fragmentation due to multiple heaps.
- Benchmark 3 tests the heap allocator's ability to reduce false cache line sharing (cache ping-ponging) on SMP hardware.

All benchmark programs are available on the project website.

4.1. Benchmark 1

We created a simple multithreaded program that invokes `malloc()` and `free()` in a loop, and times the results. To measure the effects of multithreading on heap accesses, we compare the results of running this program on a single process with the results of two processes running this program on a dual processor, and one process running this test in two threads on a dual processor. This tells us how well `malloc()` scales with multiple threads accessing the same library and heaps.

We expect that, if a `malloc()` implementation is efficient, the two *thread* run will work as hard as the two *process* run. If it's not efficient, the two process run may perform well, but the two thread run will perform badly. Typically we've found that in a poorly performing implementation, a high context switch count as a result of contention for mutexes protecting the heap and other shared resources wastes a substantial amount of kernel time.

We are also interested in the behavior of `malloc()` and the system on which it's running as we increase the number of threads past the number of physical CPUs present in the system. Many researchers conjecture that the most efficient way to run heavily loaded servers is to keep the ratio of busy threads to physical CPUs as close to 1:1 as possible. We'd like to know the penalty as the ratio increases.

For each test, the benchmark makes 10 million balanced `malloc()` and `free()` requests, for the following reasons:

1. Increasing the sample size increases the statistical significance of the average results.
2. Running the test over a longer time allows elapsed time measurements with greater precision because short timings are hard to measure precisely.
3. Start-up costs (*e.g.* library initialization) are amortized over a larger number of requests, and thus disappear into the noise.

4.2. Benchmark 2

While many multithreaded applications use and free heap-allocated objects in the same thread, network servers sometimes free heap-allocated memory in a different thread than it was allocated. Larson and Krishnan have simulated this behavior with a benchmark that we use here in a simplified form [5]. The original benchmark uses a uniform random distribution of request sizes, but we use a single request size. This simplifies the benchmark logic and the interpretation of the results [9]. Also server applications tend to use only a few request sizes [4]. Larson's goal was to create multiple stresses on allocators, but we simply want to force the allocator to leak memory.

Our single thread benchmark starts by allocating a fixed number of objects from the heap, saving their addresses in an array. The array is passed to a freshly created thread, whose job is to replace a random subset of the originally allocated objects one at a time, create a new thread, then pass the array to it and exit. Each new thread is referred to as a "round." After each run completes we record the number of minor page faults, which is proportional to the number of pages required by the allocator during the benchmark run. Linux records a minor page fault for each page allocated with `sbrk()`.

The multithread benchmark is much the same, except there are several threads concurrently replacing objects and creating new threads. In this way, threads obtain storage allocated in another thread, and must operate on this storage while the heap is under contention; these are the two conditions necessary to cause heap leakage.

We observe this indirectly with the “minor page fault” statistic returned by the `time` command.

Notice that each thread replaces a single object at a time. This fixes the total amount of heap in use during a benchmark run between mn and $m(n-1)$ objects, where m is the number of threads and n is the fixed number of pre-allocated objects. Now it is clear why we use a fixed instead of a randomly distributed object size. Because the benchmark fixes the total amount of heap storage in use at any given time, a perfect allocator should produce the same number of minor page faults for each run. Real allocators, however, show a wide variation in their final heap size because of heap leakage.

4.3. Benchmark 3

This benchmark tests how well the heap allocator places data (*i.e.*, chooses addresses for data objects) with regard to CPU cache efficiency on multiprocessor machines. If heap objects smaller than a cache line are placed in the same cache line, or if two objects overlap in a single cache line, the cache line will “ping-pong” between processor caches if the objects are modified by concurrent threads running on different processors.

Cache behavior is difficult to measure. Other studies in this area often use trace-driven simulations and synthetic allocators to discover cache behavior [3]. This is because applications can use heap objects in a variety of ways, blurring the impact and causes of cache misses and page faults.

Our goal is to create a simple, portable benchmark that indicates whether cache ping-ponging may result from heap allocated objects shared between multiple threads. To test for false sharing, we allocate n k -sized objects, where n is a number less than or equal to the number of physical processors in the system, and k is a number close to the cache line size of the system’s CPUs. We pass one allocated object to each of n threads. Each thread then writes into its object a fixed number of times. The thread writes at the front and the back of the object, in case the object overlaps cache lines. We then wait for all threads to finish, recording the elapsed time for all threads to complete. We run this test for increasing object sizes.

Note that this basic alignment test does not expose slow-downs due to cache ping-ponging of variables internal to `malloc()`, such as free list data structures or boundary tags.

5. Specific tests and results

In this section we describe our benchmark measurements, and discuss the results of each test.

	thread 1, seconds	thread 2, seconds
Avg	26.040385, s=0.013097	26.063408, s=0.006530

	process 1, seconds	process 2, seconds
Avg	23.309635, s=0.014586	23.314431, s=0.014267

TABLE 1. Average elapsed time for single heap per process versus multiple heaps per process. The two-threaded single heap test runs almost as fast as the two-process two-heap test, indicating acceptable heap contention. “s” is the standard deviation.

5.1. Benchmark 1 results and discussion

This basic test compares the performance of two threads sharing the same C library with the performance of two threads using their own separate instances of the C library. As discussed above, we hope to find out if sharing a C library (and thus “sharing” the heap) scales as well as using separate instances of the C library. We find that the shared test performs almost as well as the independent test, losing only about 10% of elapsed time. We therefore expect `malloc()` to scale well as the number of threads sharing the same C library increases.

The benchmark host for the following tests is a dual processor 200MHz Pentium Pro with 128Mb of RAM and an Intel i440FX mainboard. The operating system is Red Hat’s 5.1 Linux, which uses glibc 2.0.6¹. We replaced the 5.1 distribution’s kernel with kernel version 2.2.0-pre4. `gettimeofday()`’s resolution on this hardware is 2-3 microseconds. During the tests, the machine was at run level 5, but was otherwise quiescent.

Our first test simply runs the benchmark five times in a single thread to show heap performance when it is not contended. On our hardware, ten million allocation and release requests for 512 bytes takes an average of 23.280357 seconds, with a standard deviation of 0.005543.

The next test compares the run times of two concurrent threads that share a heap with the run times of two concurrent processes that each has their own heap. Ideally both sets of runs should be the same on a dual processor machine. Each thread or process makes 10 million allocation and free requests for 512 bytes each. The averages reported in Table 1 are over three test runs.

¹ Note that glibc 2.0 and 2.1 use nearly identical versions of `malloc()`.

	thread 1, seconds	thread 2, seconds
Avg	54.272971, s=1.146125	54.407517, s=0.833170

	process 1, seconds	process 2, seconds
Avg	6.024991, s=0.018403	6.053607, s=0.054665

TABLE 2. Average elapsed time for single heap per process versus multiple heaps per process, Solaris. The shared single heap test is almost an order of magnitude worse than the test using separate heaps.

During this test, `top` showed that both threads were using between 98% and 99.9% of both CPUs. System (kernel) time was between 0.7% and 1.1% total.

We now examine the behavior of `malloc()` as we increase the number of working threads past the number of physical CPUs in the system. In this series of tests, each thread makes 10 million allocate/free requests for an 8192 byte object. Each reported average is taken over five benchmark runs.

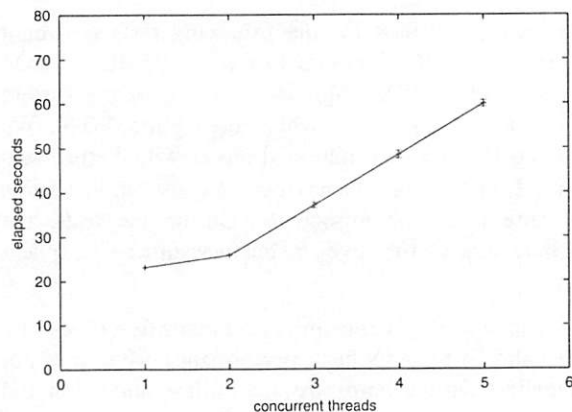


FIGURE 1. Elapsed run-time versus increasing thread count. Run-time increases with the expected slope as thread count increases, demonstrating acceptable heap contention on this hardware. Error bars indicate standard deviation.

Figure 1 shows that average elapsed time increases linearly with the number of threads at a constant slope of m/n , where n is the number of processors ($n = 2$ in our case) and m is the number of seconds for a single thread run ($m = 23$ seconds in our case).

Lastly we measure the linearity of the relationship we discovered in the last tests over a much greater number of threads. This tells us how the library scales with increasing thread count. This table contains average elapsed time measurements (in seconds) for each thread making 10 million requests of 4100 bytes each.

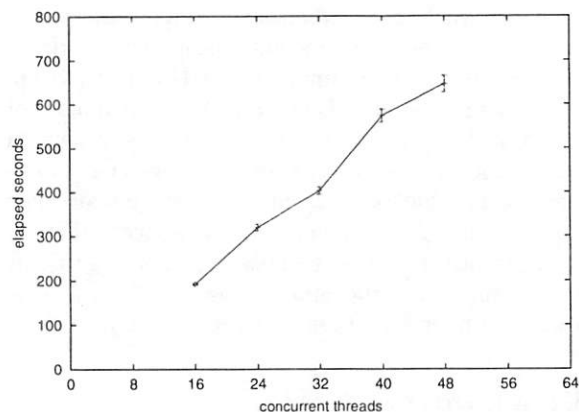


FIGURE 2. Elapsed run-time with larger thread count. On dual processor hardware, increasing thread counts have little effect on heap contention.

Figure 2 illustrates that the increase in elapsed time is fairly linear with increasing thread count, for counts much larger than the number of configured physical CPUs on the system.

Solaris tests

We ran the same series of tests on Solaris 2.6 (patch level 105181-16) running on a two CPU 400Mhz Sun Ultra AX-MP with 2G of RAM. The machine was otherwise quiescent during these runs.

Single thread timing

Single thread run time for this test averages 6.0535318 seconds, with a standard deviation of 0.0328919.

Two-thread v. Two-process

Each thread or process makes 10 million requests of 512 bytes each. Table 2 shows the results of two threads running concurrently accessing the same heap and two processes running concurrently on two independent heaps. As before, these averages were obtained over three benchmark runs.

Here we observe massive heap contention. The two-thread run is almost an order of magnitude worse than the two-process run. While the Solaris heap allocator is the fastest single thread allocator (6 second runs on 400MHZ UltraSPARC II CPUs versus 10 second runs on 500MHZ Pentium III CPUs, described below), it clearly does not scale over multiple processors.

Thread scalability

In this test, each thread makes 10 million requests of 8192 bytes. The averages are over five runs for each thread count.

	thread 1, seconds	thread 2, seconds
Avg	12.393250, s=0.000422	12.397936, s=0.000432

	process 1, seconds	process 2, seconds
Avg	10.394361, s=0.000822	10.395771, s=0.000890

TABLE 3. Average elapsed time for single heap per process versus multiple heaps per process, 4-way Linux. More processors mean slightly more heap contention. Elapsed time for shared heap test is only 20% slower than for test using separate heaps.

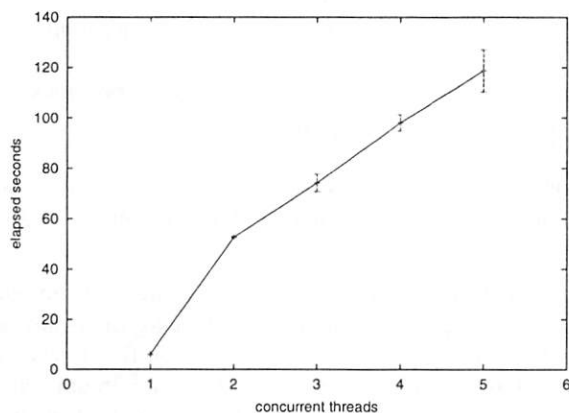


FIGURE 3. Elapsed run time versus increasing thread count, Solaris. The slope of this graph far exceeds the expected slope of 6 seconds divided by 2 processors. Lock contention is clearly a limiting factor when using a UP allocator on SMP systems.

Running five threads concurrently on dual processor Sun hardware appears to be twenty times more expensive than running a single thread.

Adding more CPUs

In this section, we present results from the same tests run on a machine with four CPUs running Linux. The hardware used in these tests is an Intel SC450NX with 512Mb of RAM and four 500MHZ Xeon Pentium III CPUs with 512Kb of L2 cache each. We loaded this machine with the Red Hat 6.1 distribution, and upgraded it's kernel to 2.2.13. It is otherwise quiescent during these tests.

Single thread timing

Single thread elapsed run time for this test averages 10.393376 seconds, with a standard deviation of 0.001243.

Run	Time in seconds
1	12.587744
2	12.587753
3	14.862689
4	12.578893
5	12.577891
6	14.844941
7	12.579065
8	12.578305
9	14.841121
10	12.576630
11	12.577823
12	14.836253
13	12.584923
14	12.584535
15	14.856683

TABLE 4. Variance in elapsed run time, 4-way Linux. Note that most runs have a 12.6 second elapsed time. Only a few have elapsed time of about 14.8, pushing the average elapsed time higher. This variance is thought to be due to allocator variables that are improperly aligned with regard to hardware caches.

Two-thread v. Two-process

As before, this test compares the elapsed time of two threads sharing a heap with the elapsed time of two processes with independent heaps.

We observe in Table 3 that there is some added expense to using multiple threads instead of multiple processes, although it is about 20% on four processor hardware. While this could be improved, it is not as bad as an order of magnitude slowdown.

Thread scalability

Each thread makes 10 million requests of 8192 bytes. Each test is run five times.

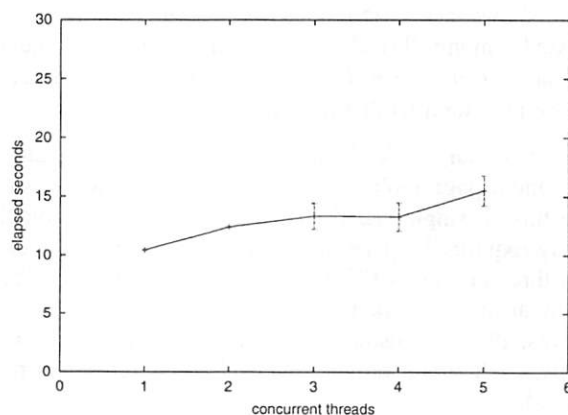


FIGURE 4. Elapsed run time versus increasing thread count, 4-way Linux. Scalability on 4 processor hardware is very good.

Notice that the average elapsed time jumps twice: once when going from one thread to multiple threads, and once when there are more threads than there are physical CPUs in the system.

A closer examination of the raw data for the three-thread run, shown in Table 4, illustrates an interesting variance in the elapsed time results. Sometimes the threads complete in 12.6 seconds, and sometimes they run for about 14.8 seconds. We see similar variances in the runs with more threads. These are likely due to what Larson refers to as *cache sloshing*. When allocator variables, such as free list pointers or condition variables, are poorly placed in memory, they cause cache lines to bounce between CPU caches. In this test, this appears to cause sporadic 20% slowdowns in the runs. We explore this phenomenon further in the section describing benchmark 3.

5.2 Benchmark 2 results and discussion

For benchmark 2, our first benchmark system is a custom-built 400MHz AMD K6-2 with 64Mb of RAM. Our system is loaded with the Red Hat 6.0 distribution, running kernel 2.2.14. During these tests, it is running a normal workstation load consisting of several xterms, a `gvim` session, and Netscape Navigator.

We choose 40 bytes as our fixed request size. Other studies have shown that network servers use only a few object sizes, and they are in the neighborhood of 40 bytes [4, 5]. Our array contains 10,000 objects per thread. We vary the number of worker thread recreations (rounds) from 1 to 8. One round means the main thread starts worker threads that stop when they are finished. Two rounds mean the main thread creates worker threads, which, when they are finished, each create a new thread, which finish and stop, and so on.

Our first test runs a single thread while increasing the number of rounds in each run. This test demonstrates that, when there is no heap contention, memory utilization shows no variation as the memory objects are passed among threads. It also indicates how much memory is consumed by a single `pthread_create()` so we may subtract that from later benchmark runs.

Based on our single thread test results, we formulate a minimum page fault count predictor as follows. A single thread, single round run that allocates a one block array requires 14 page faults. Allocating 10,000 blocks per thread requires 127.6 pages (this is 40,000 bytes for each array, and 400,000 bytes for the objects themselves, plus a constant for memory management). Finally, each round requires an additional 1.1 pages per thread.

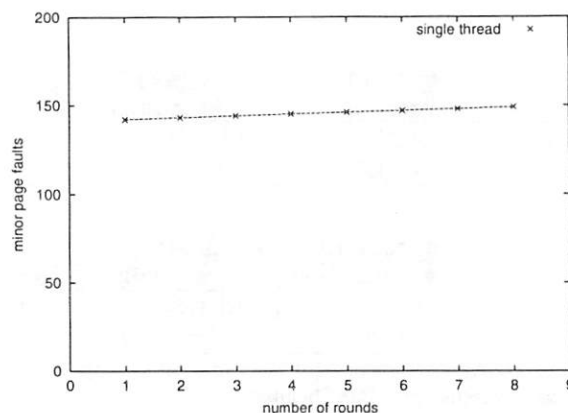


FIGURE 5. Number of rounds vs. number of minor page faults. For our single thread run, each round (creation of another thread) requires a single minor page fault over and above heap management needs. Thus each `pthread_create()` requires one extra page in the heap.

Our lower bound page fault count predictor becomes:

$$\text{mpf}_{\text{lower}} = 14 + 1.1tr + 127.6t$$

Where $\text{mpf}_{\text{lower}}$ is the lower bound minor page fault count, t is the initial number of threads and r is the number of test rounds.

The next test increases the number of threads from one to three to see how multithreading changes the behavior of the allocator. As before, the test is run five times for each fixed round count. Average, minimum, and maximum minor page fault results are reported in Figure 6.

We predict page fault count to increase by three (one for each thread) for each additional test round. In fact, at the beginning of the series shown in Figure 6, the minimum page fault count for each run series starts at 399 and increases by 3 for each additional round, as predicted. However, large variances in the number of minor page faults and larger minimum page fault counts than predicted indicate that some heap leakage occurs as round count increases.

The relative difference between the minimum and maximum page fault count in each test ranges between 25% and 50% of the measured minimum page fault count. As the number of test rounds increases, this difference becomes less, suggesting that over time, bad allocator behavior is mitigated by statistical opportunity.

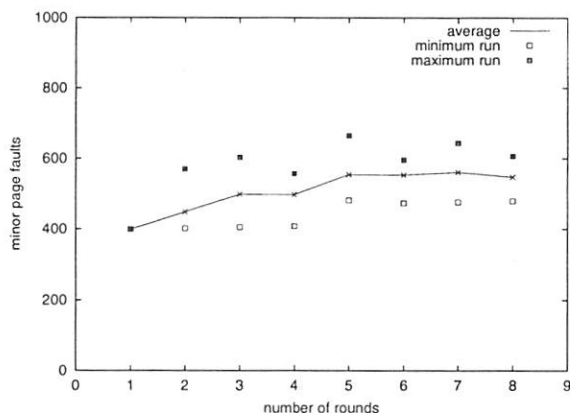


FIGURE 6. Number of rounds vs. number of minor page faults, three thread run. This run shows marked variances resulting from heap leakage. Heap size grows faster than we predicted based on what is consumed, per-thread, in the first test series.

Our final uniprocessor test increases the thread count to seven. We want to see if increasing thread count causes larger variations or if they stay the same.

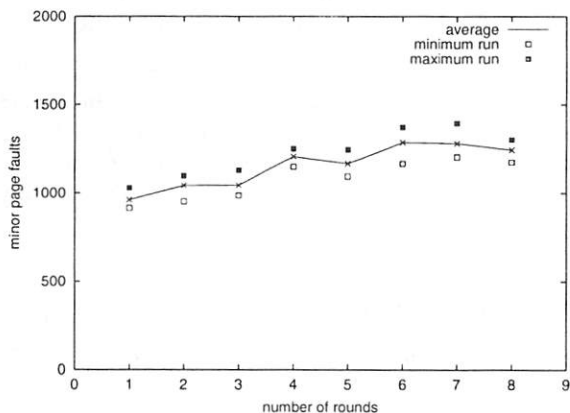


FIGURE 7. Number of rounds vs. number of minor page faults, seven thread run. Inter-run heap size variations appear to decrease with increasing thread count.

Figure 7 shows that while minimum page fault count is always larger than the predicted lower bound, the relative difference between minimum and maximum page fault counts is less in the seven thread run than in the three thread run, ranging from 9% to 18% of the minimum page fault count. This suggests that as workload increases (both thread concurrency and thread recycling) statistical behavior levels out imbalances between subheaps.

We try our seven thread run on an Intel SC450NX with 512Mb of RAM and four 500MHZ Xeon Pentium III CPUs with 512Kb of L2 cache each. We loaded this machine with the Red Hat 6.1 distribution, and upgraded it's kernel to 2.2.14. This test gives an indication

of how heap behavior changes when there is real thread concurrency. We step up the number of rounds to force behavior that might expose itself after a server application has been running over a long period.

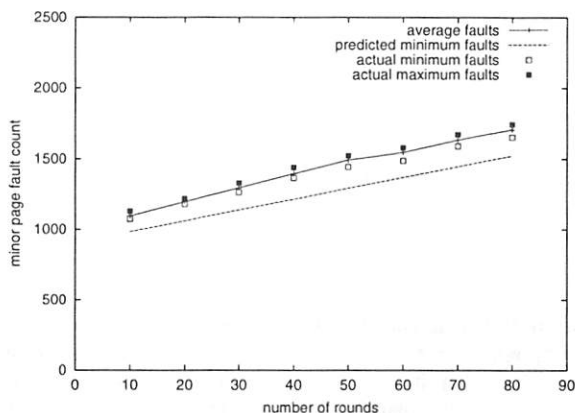


FIGURE 8. Number of rounds vs. number of minor page faults, seven threads on four CPUs. The slope of the actual page fault count follows our predictor function, so the allocator is behaving reasonably well.

Even on a four processor server, `malloc()` on Linux appears to behave well. The minor page fault count averaged over five runs increases with approximately the same slope as our predictor function. The actual values are offset from the predicted values by nearly a constant. While there is some unpredictability in the amount of heap required for each test run, the heap doesn't appear to grow in an unbounded manner as the amount of heap activity increases.

5.3 Benchmark 3 results and discussion

Benchmark 3 was run on our Intel SC450NX server containing four 500MHZ Pentium III CPUs, each with 512K of Level 2 cache, a typical SMP server configuration. The benchmark starts one or more threads that attempt to write into a heap-allocated object 100 million times. A single thread running the benchmark on this hardware completes in 2.102 to 2.103 seconds. This result is independent of object size because we write only a single byte at the front and back of each object.

Figure 9 compares the elapsed time of the benchmark against properly cache-aligned objects versus the same benchmark with arbitrarily aligned objects. Two threads compete with each other in this test. The test runs on object sizes between three and 52 bytes in order to vary the alignment of the objects with respect to hardware cache lines.

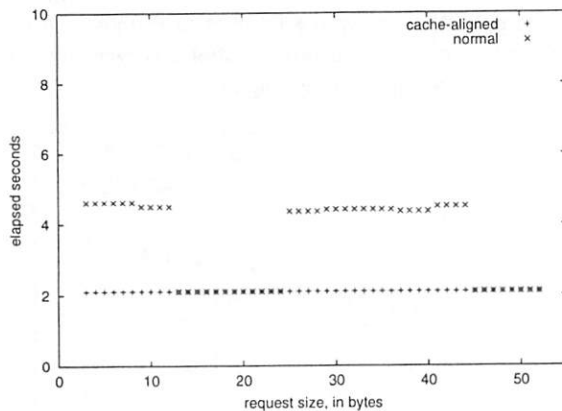


FIGURE 9. Cache sharing between two threads. Cache line sharing results in twice the amount of elapsed time per write operation. From this test, it is clear that heap allocators that prevent cache line sharing can boost application performance.

Here we clearly see that cache line sharing between two CPUs can cause a slow-down of more than half when the object is being concurrently modified. In other words, if two objects happen to overlap in a cache line, it can take more than twice as long for writes into each object to complete. Figure 10 shows the same test with the thread count increased from two to three. While each object's cache line is potentially shared between only two CPUs at a time, there is still a large penalty.

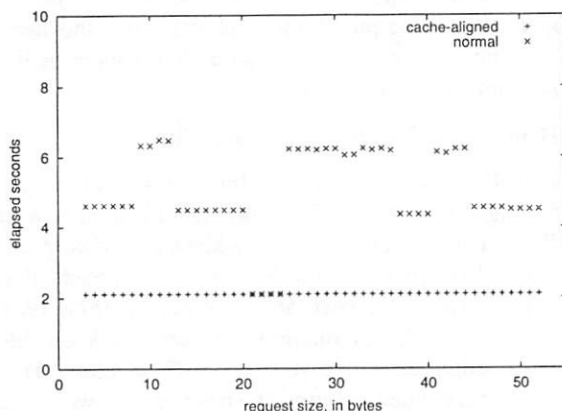


FIGURE 10. Cache sharing between three threads. This test shows the impact of false cache sharing among three processors due to improper heap object alignment.

Figure 11 depicts the same test with thread count increased from three to four.

Four threads modifying independent cache lines on this hardware can run almost as fast as a single thread. As soon as cache line sharing occurs, write performance is greatly reduced, sometimes by as much as a factor of four.

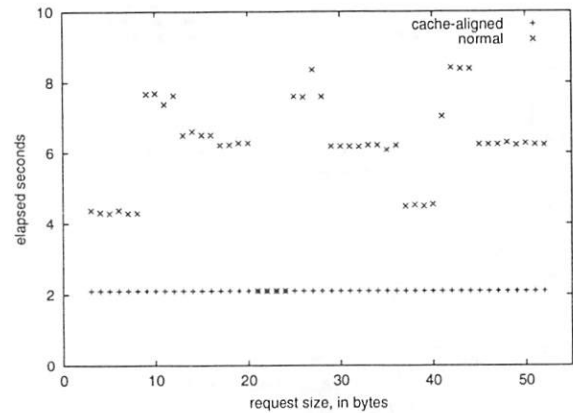


FIGURE 11. Cache sharing between four threads. This test shows large elapsed time variances as well as substantial slowdowns in write operations.

The precise results obtained in the tests that used normally aligned objects are not repeatable because the addresses of objects returned by `malloc()` are somewhat nondeterministic. However we observe that some of the time objects are aligned in such a way that false sharing occurs and application performance suffers.

While it is common wisdom that cache line sharing can affect application behavior, these tests demonstrate conclusively that this impact can be substantial. We note that this test artificially highlights alignment problems. Real applications will likely not be as profoundly affected by object misalignment.

6. Conclusions and Future Work

Our tests show that the `malloc()` implementation used in glibc 2.0 and 2.1 handles increasing numbers of threads effectively while adding little overhead, even for a large number of threads. We find expected performance curves as offered load increased. Other studies, such as Berger and Blumofe, that have increased the number of CPUs in their systems far past four have found that glibc `malloc()`'s performance degrades for large numbers of CPUs [1]. However for the two- and four- CPU systems commonly used in today's server farms, glibc's `malloc()` performs acceptably well.

Many allocators cause unbounded heap growth when an application allocates objects in one thread and releases them in another. Our benchmarks show that, even under contention, glibc's allocator becomes less efficient, but doesn't show pathological heap growth.

We also note potential slow-downs that can result from poor alignment of heap objects with respect to the Level 1 CPU cache line size. These slow-downs can be mitigated either by careful application design or by accepting a heap allocator that aligns objects automati-

cally to cache line boundaries, and thereby increases heap fragmentation. Application developers might make use of two different allocation mechanisms: one for thread-private objects that provides tight alignment to reduce fragmentation and memory utilization, and one for objects that may be shared among threads that uses cache-aware alignment to reduce false cache sharing.

In the future, we plan to run tests that include two important areas not considered in this paper. Heap allocator latency should show little or no change as network servers remain up over time. We plan to create a benchmark to measure latency changes over server uptime. We also plan to test our assumptions about the allocation patterns of large-scale network servers by instrumenting heavily used servers to generate trace data.

Wilson, Zorn, and many others have spent considerable effort optimizing the basic algorithms for single threaded allocation. However, a close examination of the performance relationship between the C library's memory allocator and OS primitives such as mutexes, `mmap()`, and `sbrk()` might show some interesting trade-offs.

Finally, we plan to examine the performance and scalability of kernel-level memory allocators with these same criteria in mind. The kernel's slab allocator uses a single spin lock in each slab cache to control access among multiple threads. This has the same performance implications as using a single spin lock at the user level.

6.1. Acknowledgements

The authors thank Emery Berger for his contribution to our efforts, and thank our reviewers for their cogent comments. Special thanks go to Dr. Charles Antonelli, Intel Corporation, Seth Meyer, and Hans C. Masing for equipment loans.

7. References

- [1] E. Berger, R. Blumofe, "Hoard: A Fast, Scalable, and Memory-Efficient Allocator for Shared-Memory Multiprocessors," The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-99-22. September 1999.
- [2] W. Gloger, "Dynamic memory allocator implementations in Linux system libraries," www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html.
- [3] D. Grunwald, B. Zorn, and R. Henderson, "Improving the Cache Locality of Memory Allocation," *SIGPLAN Conference on Programming Language Design and Implementation*, June 1993.

- [4] M. S. Johnstone and P. R. Wilson, "The Memory Fragmentation Problem: Solved?" *Proceedings of the First International Symposium on Memory Management*, ACM Press, October 1998.
- [5] P. A. Larson and M. Krishnan, "Memory Allocation for Long-Running Server Applications," *Proceedings of the First International Symposium on Memory Management*, ACM Press, October 1998.
- [6] D. Lea, "A Memory Allocator," *unix/mail*, December 1996. See also g.oswego.edu/dl/html/malloc.html.
- [7] P. Wilson, M. Johnstone, M. Neely and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *Proceedings of the 1995 International Workshop on Memory Management*, Springer LNCS, 1995.
- [8] B. Zorn and D. Grunwald, "Empirical measurements of six allocation-intensive C programs," *ACM SIGPLAN notices*, 27(12): 71-80, 1992.
- [9] B. Zorn and D. Grunwald, "Evaluating Models of Memory Allocation," *ACM Transactions on Modeling and Computer Simulation*, 4(1): 107-131, 1994.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *login:*, the Association's magazine, published eight-ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association:

Earthlink Network	Microsoft Research	Server/Workstation Expert
Greenberg News Networks/MedCast Networks	MKS, Inc.	Sun Microsystems, Inc.
JSB Software Technologies	Motorola Australia Software Centre	Sybase, Inc.
Lucent Technologies	Nimrod AS	Syntax, Inc.
Macmillan Computer Publishing, USA	O'Reilly & Associates Inc.	UUNET Technologies, Inc.
	Performance Computing	Web Publishing, Inc.
	Sendmail, Inc.	WITSEC, Inc.

Supporting Members of SAGE:

Collective Technologies	Mentor Graphics Corp.	RIPE NCC
Deer Run Associates	Microsoft Research	SysAdmin Magazine
Electric Lightwave, Inc.	MindSource Software Engineers	Unix Guru Universe
ESM Services, Inc.	Motorola Australia Software Centre	
GNAC, Inc.	New Riders Press	
Macmillan Computer Publishing, USA	O'Reilly & Associates Inc.	
	Remedy Corporation	

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: office@usenix.org.

